Mitigator: Privacy policy compliance using Intel SGX

by

Miti Mazmudar

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Privacy policies have been known to be hard to read and understand by internet users and yet users are obliged to accept these one-sided terms of usage of their data before they can effectively use websites. Although research has been conducted into alternative representations of privacy policies, it does not consider whether the website provider actually adheres to the data handling practices outlined in the privacy policy. However, there has been significant research towards achieving compliance of internal processing systems to access control policies that capture some aspects of privacy policies, such as those related to confidentiality of collected information, the time period of its retention, and its disclosure to third parties. Apart from the fact that these access control policies may not be designed to be translatable to machine-readable or simplified text policies, such systems suffer from two related drawbacks: first, they assume a large trusted computing base (TCB) and in particular, the operating system is included within their TCB. Secondly, as they are only aimed at achieving compliance of different *internal* data processing systems to these access control policies, they do not seek to provide users of any proof of a compliant system.

On the other hand, trusted hardware seeks to reduce the TCB on a remote machine that a user needs to trust in order to run a program and obtain its results. Trusted hardware platforms provide two novel security properties: they disallow a malicious operating system from learning secrets from the program state and secondly, they allow the user to verify that the OS has not modified the program before or while running it, as long as the user trusts the hardware platform. Our goal is to design an architecture that uses an underlying trusted hardware platform to run a program, named the *decryptor*, that only hands users' data to a target program that has been determined to be compliant with a privacy policy model. As both of these programs are run on a trusted hardware platform, users can verify that the *decryptor* is indeed the correct, unmodified program. Most importantly, in our architecture, we provide trustworthy information about the *verifier* program used on the server side to a client program such that it can ensure that the target program has been checked for compliance with a privacy policy model by a valid *verifier* program. Such a verifier program should be made open-sourced so that it can be checked by experts. Our second contribution lies in implementing this architecture on the Intel SGX hardware platform, using a shim layer, namely the Graphene-SGX library. Finally, we also evaluate our system for its efficiency and find that it has a very small overhead in comparison with a setup that does not provide such guarantees.

Acknowledgements

I would like to thank my supervisor, Ian Goldberg, for introducing me to the world of privacy-enhancing technologies, for his constant support throughout my program, and in guiding me through the research process. I would also like to thank my committee members Xi He and Ali Mashtizadeh for their feedback on this thesis. I have grown significantly in personal and professional ways in my time so far in the CrySP lab. My thanks go to all members of the CrySP lab for sharing their knowledge, experiences, and in general, for all the good times that we have had! Huge thanks go to Cecylia, Justin, Nikita, Bailey, Brittany, Navid, Erinn, Sarah, Sajin, Masoumeh, and Nik for their unwavering support in helping me write and finish this thesis. This work benefitted from the use of the CrySP RIPPLE Facility at the University of Waterloo.

Dedication

This is dedicated to my parents for their continuous support.

Table of Contents

List of Tables ix

List of Figures x

1 Introduction 1

1.1 Contributions and roadmap 4

2 Background 5

2.1 Modelling natural language privacy policies 5

2.2 Detecting privacy violations in source code 7

2.2.1 Analyzing smartphone apps for privacy violations 8

2.2.2 Source-code analysis of web applications 9

2.3 Trusted hardware platforms 11

2.3.1 Supporting complex applications in Intel SGX enclaves 14

2.4 Secure browser-to-user interfaces 18

2.5 Summary 20

3 Related work 21

3.1 Trust-based compliance — internet seals 21

3.2 P3P: Machine-readable privacy policy models and the need for ensuring compliance 24

3.3 Systems for internal compliance . . . 25
3.4 Trusted hardware platform-based compliance systems . . . 26
3.5 Summary . . . 29

4 Model and design **30**
4.1 Threat model . . . 34
4.2 System design . . . 35
4.2.1 Verification stage . . . 36
4.2.2 Deployment stage . . . 40
4.2.3 Runtime . . . 41
4.2.4 Verifying the integrity of enclaves from the client . . . 44
4.3 Security analysis . . . 46
4.3.1 TLS termination . . . 47
4.3.2 Sealing . . . 48
4.3.3 Security while updating enclaves . . . 50
4.4 Design choices . . . 52
4.4.1 Distributing enclaves across server-side machines . . . 52
4.4.2 Choice of sealing key input . . . 52
4.4.3 Comparison to related work . . . 53
4.5 Summary . . . 55

5 Implementation **56**
5.1 Decryptor enclave . . . 56
5.2 Adapting the Graphene-SGX platform . . . 65
5.3 Verifier enclave . . . 65
5.4 Adapting a source code analysis tool: Pixy . . . 71
5.5 Target enclave—PHP Extension . . . 72
5.6 Browser extension . . . 76
5.7 Summary . . . 79

6 Performance Evaluation **80**

6.1 Setup . . . 81

6.2 Instrumentation . . . 82

6.3 Server-side latency . . . 83

6.4 End-to-end latency . . . 90

6.5 Summary . . . 93

7 Future work **94**

8 Conclusion **96**

References **97**

APPENDICES **107**

A Attestation in Intel SGX **108**

A.1 Local attestation . . . 108

A.2 Remote attestation . . . 116

B Sealing-relevant algorithms **119**

List of Tables

2.1 Comparison of platforms that support running complex applications within Intel SGX 17

5.1 Decryptor enclave internal state 57

List of Figures

4.1 A high-level block diagram of Mitigator. 36
4.2 Mitigator verification stage - verifier block diagram 37
4.3 Mitigator deployment stage 39
4.4 Mitigator runtime stage 42

6.1 Measurement of server-side latency from the localhost interface as the size of a form field increases 85
6.2 Server-side latencies from the localhost interface as the number of form fields increases 86
6.3 Measurement of server-side latency from the remote machine as the size of a form field increases 88
6.4 Server-side latencies from the remote machine as the number of form fields increases 89
6.5 Distribution of time in median runs as the length of a single form field sent in the request varies 91
6.6 Distribution of time in median runs as the number of form fields sent in the request varies 92

A.1 One-sided local attestation - report generation and verification 109
A.2 Two-sided local attestation 111
A.3 Remote attestation 114
A.4 Remote attestation— enclave to quoting enclave interactions 115

Chapter 1

Introduction

Internet users have been habituated to click on a checkbox when they register an account on a website. Doing so implies that they acknowledge that they have read and understood a privacy policy, which details how the website provider processes and disseminates their data and possibly what forms of metadata are collected. However, these privacy policies are lengthy and written with legalese-filled jargon, as McDonald and Cranor [MC08] found. Users cannot be expected to have read and understood such a text that is aimed at a specialist audience, namely lawyers, thereby leading users to a Kafkaesque interface.

Towards this end, Cranor et al. [CLM+02] proposed a machine-readable XML privacy policy format known as the Platform for Privacy Preferences Project (P3P), which was set as a W3C standard. P3P was intended in order to support users in specifying certain simple policies. These policies encapsulated choices on the collection, dissemination, and retention of users' data and metadata. The user's P3P policy would be matched against that of the website that they visited for compatability and users would be alerted to incompatible sites; that is, ones whose P3P policy involved the collection of more data or metadata than the user wishes or which retained data longer than the user desired. Despite these desirable properties, P3P had limited deployment and remains as an obsolete W3C standard at the time of this writing. The current P3P W3C standard document [CLM+02] reports that website admins often used P3P policies that were not reflective of their actual practices, as there were no consequences to doing so. We can see that simply trusting the server-side code on website providers' machines to handle data as per the privacy policy does not work. Such a trust assumption also applies to companies that claim to provide seals or certifications on the privacy practices of website providers. Any audits or checks that the company performs on the website providers cannot be proven to a user: users simply have to trust that this company did indeed perform some checks on the website provider's infrastructure.

Indeed, this trust assumption is widely misplaced in today's internet: for instance, in the Cambridge Analytica scandal [Mer18], a smartphone app obtained users' personal information and detailed psychological profiles, and on the server side, linked each user's profile with information from Facebook profiles of their friends, to result in a large repository of personal information and preferences of the general population that would then be exploited for showing microtargeted ads for political campaigns. On account of such poor data handling practices, it is reasonable for a user to not trust web application code, including the server-side code, in terms of achieving compliance with the goals stated in the privacy policy. For example, one server-side script may output users' personal information to a file that is then sent to a third party. This example is not unrealistic: data that users enter into websites or apps has been known to be sold or transmitted to third parties [SWH+16] [ZWZ+17] and data brokers [RBO+14]. It therefore illustrates that a website user may potentially consider any server-side website code as malicious or privacy invasive.

Additionally, recent literature on achieving compliance of source code with internal or external policies [SGD+14] [EMV+16] trusts the website providers' machines, possibly including the operating system itself, to be completely secure or free of any vulnerable code. Vulnerable components in web applications, such as servers, libraries and even operating systems, remain a major risk factor for the security of web applications [Pod17] as well as a major root cause of data breaches today [Whi19]. Although patching vulnerable code is the recommended approach and is orthogonal to our work, the existence of any vulnerable code can lead to an exploit. Such an exploit may result in the dissemination of users' data or its usage for an uintentional purpose and thus, the problem of vulnerable server-side machines cannot be ignored. Given that website providers could possibly be running vulnerable operating systems and applications on their machines, they ought to confirm compliance of their source code with privacy policies.

A few years after the deployment of P3P, Ashley et al. [AHK+03] proposed the fine-grained Enterprise Authorization Policy Language (EPAL). EPAL was an access control langauge to assist enterprises develop an enterprise-wide internal privacy policy, exploiting common specifics of data processing implementations throughout enterprises. Systems that use access control mechanisms to encapsulate privacy preferences on the confidentiality and dissemination of data, as well as to implement these within organizations, continue to be developed today. For instance, Sen et al. [SGD+14] develop a first-order logic to encapsulate modern privacy policies, named Legalease, and bootstrap an information flow analysis tool, Grok, to implement an internal access control mechanism based on this logic. Thoth by Elnikety et al. [EMV+16] improves on Grok in that it enables implementing user-specified privacy settings through an access control mechanism, using a reference monitor within the kernel of server-side machines. Thoth seeks to reduce the trusted computing base from all applications that operate on users' data, to simply the operating system, its reference monitor, and a kernel module. While developing mechanisms to

implement privacy policies, whether user-specified or established internally, is important, a common downside of such systems is that they have not been designed with a view towards proving or demonstrating their efficacy to users.

We improve on existing work in fulfilling the two aforementioned requirements: namely, we check compliance of server side code to a privacy policy in the face of an untrusted operating system on the remote machine and demonstrate a proof of a successful compliance check to users. This problem is conceptually similar to the secure remote computation problem, which is to run a given program on a remote computer and obtain its results, in the face of an untrusted remote operating system that may attempt to modify the program or learn secrets based on the program state. The security research community has focused on ways to solve the secure remote computation problem for many years; trusted hardware platforms are one proposed mechanism. These platforms involve establishing trust in a small part of the CPU in the remote host machine, while treating the rest of the hardware and the operating system of the remote host as untrusted and compromised. This part consists of a set of hardware-infused keys and microcode. Traditionally, the trusted hardware was in the form of a separate physical module [BJG+18], but it has recently been integrated into CPU chips. Trusting this hardware allows a client to run a program on the remote host within the trusted hardware platform, while obtaining confidentiality of the program state and integrity of the program. A client can also conduct an attestation protocol to verify that the desired program, and not a modified version of it, is being run on the remote host. Korba et al. [KK03] propose a design based on trusted hardware platforms, which have been marketed as Digital Rights Management (DRM) tools, towards the problem of compliance of source code with privacy policies.

Our main insight lies in using a trusted hardware platform to narrow down the trusted computing base (TCB) of a compliance mechanism running on the remote, server-side machines of a website provider. Effectively, we narrow down the TCB from one that includes all of the operating system and hardware, as in Thoth, to only a trusted hardware chip and relevant supporting libraries. Our design is applicable to websites run on any trusted hardware platform; for our proof-of-concept implementation, we use the Intel Software Guard Extensions (SGX) platform as our trusted hardware platform. Additionally, we use the attestation property of this platform to assist the user in verifying that the program that performs the compliance check has not been modified. We therefore overcome the drawback of internal compliance assurance systems by designing our system such that the user can verify the compliance check. This work aims to prove the following thesis statement:

It is possible to build a system that can provide a verifiable guarantee to a user that the personal data that they submit to a website can only be processed by code that has been verified to be compliant with the website's privacy policy. The guarantee is verifiable in that the user can verify the integrity of the code that does this compliance check.

1.1 Contributions and roadmap

We start with a discussion of the basics of trusted hardware platforms, privacy policy models, and website source code analysis tools in Chapter 2. This provides us with the foundation to demonstrate how we achieve the above thesis statement through the following four contributions that are substantiated in the subsequent chapters.

1. We discuss the advantages of our approach as compared to other related work within the domain of achieving compliance of source code with a privacy policy model (Chapter 3).

2. We provide a cryptographic design that securely uses underlying trusted hardware platform primitives to provide the guarantee claimed in the thesis statement (Chapter 4).

3. We describe a proof-of-concept implementation of the above cryptographic design on the Intel SGX trusted hardware platform (Chapter 5).

4. We evaluate our proof-of-concept implementation, in terms of overheads on an identical setup that runs on an untrusted operating system and does not provide the above privacy guarantee. We perform this evaluation in Chapter 6.

Our proof-of-concept implementation is named Mitigator. Future versions of Mitigator are intended to be deployed by organizations and individuals who wish to assure their websites' users that they process the users' data in a privacy-compliant manner. These organizations may have differing incentives for doing so, such as compliance with legislation, expanding their user base to privacy-conscious individuals, or simply to reassure their users. Mitigator provides the above guarantee even in the face of malicious programs on the organization's machines and vulnerable operating systems.

Chapter 2

Background

In this chapter, we lay out concepts and related work in literature that are used in our design and implementation. We start with related work in forming models out of natural language privacy policies in Section 2.1, as any privacy policy text to be used with Mitigator needs to be converted into an appropriate model that can be used for checking compliance of the source code. We then discuss approaches and tools that can be used for the latter; that is, to check the source code for compliance or alternatively, for privacy violations, in Section 2.2. We proceed to describe trusted hardware platforms and the guarantees provided by a major trusted hardware platform that we use within Mitigator's proof-of-concept implementation, in Section 2.3. In this section, we also discuss recent literature on systems that support complex applications on these platforms, one of which is used to develop Mitigator. Finally, Mitigator needs to ensure that users' plaintext data is encrypted to a remote host running on a trusted hardware platform, without any dynamic webpage scripts or other extensions obtaining the plaintext data or tricking users into believing that it is being encrypted when it is not. We thus discuss secure user-to-browser interfaces in Section 2.4.

2.1 Modelling natural language privacy policies

Many techniques from the fields of requirements engineering and natural language processing have been applied to extract various models out of textual privacy policies. Breaux and Schuab [BS14] sought to decrease the cost of experts manually annotating privacy policies and to do so, they formulated and validated a partially crowdsourced system. A crowdsourced worker would obtain a paragraph of a particular privacy policy and would be asked to identify action

verbs from a known list. If they identified certain action verbs, they would then be asked to specify information types on which the action verbs apply, the purposes for conducting this action and in case of transfers of data, intended sources and targets. A crowdsourced workers' annotation result would be considered for renumeration as long as it agreed with a minimum threshold of other workers.

In contrast with Breaux and Schaub's crowdsourcing approach to annotate privacy policies, Wilson et al. [WSD+16] engaged experts in a discussion process over encodings of a small set of privacy policies. The experts established a set of ten data categories, with each data category containing mandatory[1] and optional attribute-value pairs, known as data practices, to further specify it. Each paragraph in the rest of the dataset of privacy policies would be annotated with data categories and data practices from the above set by skilled annotators, using a grounded theory approach, wherein an annotation that at least two annotators agreed on would be considered as ground truth.

ToS;DR [JBR+12a] is an online crowd sourcing-based platform that was created by experts to allow internet users to grade privacy policies among different topics, culminating in a final grade or class for each privacy policy. Interestingly, these topics [JBR+12b] are more general than the ones generated through the aforementioned expert or controlled crowdsourcing processes. They include grades for specification of applicable jurisdictions, data portability, account suspension or censorship, transfer of data to law enforcement and government requests, ownership or copyright licenses, right to leave the service, and so on. Zimmeck and Bellovin [ZB14] propose a browser extension named Privee that retrieves the ToS;DR grade for the URL that a user has visited. In absence of a ToS;DR grade, it automatically provides a score based on practices related to the following six categories: collection of data, encryption, limited retention, profiling, tracking through ads, and disclosure of personal information through ads. When it is first loaded, the extension is initialized with a small sample set of training policies and it then sets up a binary ML classifier to determine whether the given privacy policy allows an operation for each of the above six categories or not. We can thus see that Privee conducts a coarse-grained analysis of the privacy policy: Privee's grade is not influenced by the types of information under consideration in the privacy policy.

Harkous et al. [HFL+18] present a browser extension named Polisis that conducts a scalable, significantly more fine-grained analysis of websites' privacy policies. Polisis preprocesses privacy policies into semantically complete segments and then passes these segments as inputs to a set of heirarchical multi-label classifiers that predicts the probability that a given high-level data

[1]Mandatory attributes for data practices were also obtained as an output of the expert discussion process. Each data practice has a mandatory attribute whose value is the text which it corresponds to or condenses. This helps analysis by other human experts or systems.

category label in Wilson et al.'s [WSD+16] model is present in the input. If a data category label is more likely than not to be present in the segmented input, then for that input, a lower-level classifier predicts the probability of each possible value for each attribute in the data category. About half of Wilson et al.'s [WSD+16] annotated privacy policies are used to train these classifiers, whereas the rest are used to test them. The authors report a high average F1 score of 0.84 across all data categories. Harkous et al.'s classifiers could be made to support synthesizing other models of the privacy policy. For instance, Sen et al.'s [SGD+14] first-order logic representation of privacy policies, named Legalease, uses finite lattices to represent values — Polisis can be extended to automatically generate Legalease representations.

Harkous et al. also design a bot named Pribot to automatically answer free-form queries for privacy policies; these are particularly useful for conveying privacy information to users on devices without screens, such as smart IoT devices. They test it against a curated dataset of Twitter questions that were aimed at companies' official data handles and contained privacy-relevant terms. For the ground truth, the authors followed a grounded theory approach to annotate data category labels in the corresponding privacy policies for each of the questions. They found that the annotators' label was present in the top of the list of output data category labels for 68% of all Twitter questions and it was in the top 4 elements of the list for 87% of all questions. Harkous et al. also design privacy icons based on Polisis: a label with a red, yellow, or green colour is assigned based on whether the privacy policy provides no choice, opt-out choice, or, opt-in choice for a specific category and an attribute in that category, such as first-pary collection for advertising purposes.

In general, Mitigator needs a model of the privacy policy that can be used to configure a source code analysis tool to check the source code. Polisis can be adapted to form a model of the privacy policy text that contains information types and operations at a sufficient granularity to configure the source code analysis tool. Mitigator's signal of compliance can be used to augment these privacy icons, as these icons do not consider whether the source code is compliant with the privacy policy.

2.2 Detecting privacy violations in source code

We first analyze two recent works that form models of smartphone apps' privacy policies and compare the source code of these apps against these models, in order to identify privacy violations. We discuss possible extensions of these systems in the context of Mitigator and the repercussions of the authors' findings for website privacy policies and privacy violations. We then proceed to the context of websites and discuss back-end source code analysis tools that

have been used to detect security vulnerabilities within websites. Finally, we discuss the observation that Mitigator's source code compliance check can be formulated in terms of a problem of source code analysis.

2.2.1 Analyzing smartphone apps for privacy violations

Slavin et al. [SWH+16] identify apps that send device-specific information or directly collected information across the network when the privacy policy either does not specify it (a "strong" violation in the authors' terms) or specifies a generic term for the given information type (a "weak" violation). They first form two sets of annotations, wherein a given term is mapped into one or more privacy-relevant words by expert human annotators; Breaux and Schuab's [BS14] framework for crowdsourcing experts' annotations is used to facilitate this. Expert annotators form the first set by annotating each method within the Android SDK, based on the natural language description of the method and its input and output parameters. They form the second set of annotations by annotating a small number of mobile privacy policies for device-specific information and for directly collected information. When deduplicated, this second set gives rise to a small set of 368 privacy-relevant information types. Slavin et al.'s novel contribution lies in forming an ontology that relates these sets of annotation terms through an inclusion relation as follows: each Android SDK method forms a leaf in the ontology and its annotations and the privacy policy's annotations form non-leaf nodes. In other words, their ontology allows forming a list of natural language terms, going from most specific terms to most general ones, that can be used to describe data that an Android SDK method operates on. As browser extensions expose a limited API for dynamic Javascript and HTML code, this technique can be extended to the website domain to identify whether client-side code running within a browser violates its privacy policy.

Using an existing dynamic taint analysis tool and a pre-defined list of taint sink methods that send data across the network, the authors automatically analyze each app to identify whether it contains Android SDK methods that send data over the network and if so, determine the information types sent over the network. Then, for each leaf node in the ontology, which identifies an Android SDK method that sends data in the code, they form a set of all nodes in paths to the root of the ontology. If none of the nodes in this set are present in an automated annotation of the privacy policy, then effectively, the privacy policy does not list that this information type, whether in a general or a specific form, is being sent across the network and this is taken to constitute a strong violation for that leaf-node method. Whereas, if the privacy policy annotation contains a non-leaf node other than the first non-leaf node in this path, then the privacy policy effectively contains a generalized description of the information type that is sent across the network and it is taken to have a weak violation for that method.

Zimmeck et al. [ZWZ+17] form a set of requirements that the content of privacy policies should meet, if they were subject to privacy-relevant laws in the United States. These requirements include the following: presenting a privacy policy, informing users of changes in this policy, of any personally identifiable data that is being collected and/or possibly shared with third parties, as well as including mechanisms to access, edit and delete their information. The authors use Wilson et al.'s [WSD+16] set of annotated privacy policies for identifying terms for various privacy-relevant operations. They then use a combination of natural language processing and machine learning techniques to form a model of the privacy policy text with respect to the above requirements. They perform static analysis of a test set and a full set of popular Android applications, to identify which Android system API calls are being called by each application and also determine the names of any third-party libraries[2] that perform these calls.

The authors find that a significant percentage of apps without privacy policies collect (87%) and share (62%) contact information with third-party libraries. 50% and 63% of apps that do present privacy policies are found to be inconsistent in stating that they collect and share contact information, whereas much smaller, but non-negligible fractions of such apps are inconsistent in stating the collection and sharing of location information (41% and 17%). Zimmeck et al. and Slavin et al.'s experimental results provide evidence for inconsistencies between smartphone privacy policies and their app implementations. They provide us with reasonable grounds to believe that such inconsistencies would also exist for desktop versions of websites. As we mentioned above, browser extensions are another relevant context wherein Slavin et al. and Zimmeck et al.'s source code annotation and analysis techniques can be applied to identify non-compliant Javascript code. Mitigator focuses on the processing of users' data on server-side machines using back-end languages, as opposed to client-side machines. In fact, even though back-end languages do not explicitly deal with as many explicit APIs to collect different information types as smartphone SDKs do, these approaches can be extended to the server side as well, by annotating the source code with data type that it is processing, as was done by Sen et al. [SGD+14] for Grok. Sen et al. and Slavin et al. use dynamic source code analysis tools, whereas Zimmeck et al. use a static source code analysis tool. Mitigator has been currently designed to conduct static source code analysis of back-end website code to provide gurantees of compliance.

2.2.2 Source-code analysis of web applications

We proceed to discussing security vulnerabilities that can be detected through the use of taint analysis tools. Security vulnerabilities such as SQL injection and XSS attacks occur due to a

[2]They manually verify that correct usage of these libraries' relevant API calls does result in sharing of personally identifiable data, such as contacts or location data.

lack of validation of user-specified input. Specifically, an XSS attack occurs when unvalidated user-specified inputs find their way into the output of any server or client-side script, whereas SQL injection attacks occur when user-specified input is not validated before it is entered as a part of an SQL query. Information flow analysis has been used by security researchers to analyse server-side PHP scripts to detect such vulnerabilities. Jovanovic et al. [JKK06] propose Pixy, an open-source PHP taint analysis tool for automatically identifying vulnerabilities due to a lack of sanitization, including SQL injection and XSS attacks, in server-side PHP code.

We briefly describe how taint analysis of back-end scripts, such as PHP scripts, can be used to find code vulnerable to XSS attacks. Any PHP script can obtain data sent to a webpage hosting it, through any HTTP request method, including GET and POST, in the form of arrays of values. Cookies and session values are also accessible to all PHP scripts in a similar way. Pixy marks these array values as *tainted* and it allows developers to configure a list of *sanitization* functions that remove potentially malicious characters for SQL queries (SQL code injection) or for Javascript code (XSS attacks), as well as a blacklist of output functions that print passed inputs into HTML and could potentially be exploited for the above attacks. Taint analysis is used to ascertain whether tainted input arrays are passed to an output function, without being passed through XSS or SQL sanitization functions. If they are, then the source code is reported to be vulnerable to the respective attacks. Pixy runs on individual PHP source code files as inputs and outputs a binary result, corresponding to whether the code is vulnerable or not.

Most recently, Backes et al. [BRS+17] proposed PHP-Joern to conduct scalable, automated analysis of PHP scripts to detect a wider range of security vulnerabilities, such as control flow-related vulnerabilities. They construct novel graphs that combine information from control flow, data flow analyses and abstract syntax trees of the PHP script and store these in graph databases. Developers are to define vulnerabilities in terms of graph traversal problems over these graphs and query these graph databases for vulnerable nodes. Owing to its simplicity and ease of integration into Mitigator's implementation as a simple stand-alone unit, we currently use Pixy within Mitigator to analyze PHP source code files. However, other taint analysis tools can be extended to support a compliance analysis, as we discuss in Chapter 5.

We have seen the use of taint analysis tools to determine privacy policy violations in Slavin et al. and Zimmeck et al.'s work above; for instance, the flow of users' data to functions that send data across networks could constitute a privacy policy violation, depending on whether the privacy policy model includes information about this transfer or not. We modify Pixy to support identifying a simple privacy-relevant violation: passing users' data unencrypted to files; that is, not encrypting users' data at rest. Our implementation currently uses Pixy's configuration setup for listing sanitizing and output functions as a simplified a privacy policy model. This is not a limitation of our design; as we mentioned previously, other tools like Harkous et al.'s [HFL+18]

Polisis could be used to automatically extract relevant models that can be used to configure source code analysis tools for compliance checking.

2.3 Trusted hardware platforms

Trusted hardware platforms have been designed to solve the secure remote computation problem: a developer wishes to run a program on a remote untrusted machine and obtain an assurance of the confidentiality of the program state and integrity of the program. Two prominent trusted hardware platforms include Intel's Software Guards Extension (SGX) platform [Int19b] and AMD's Secure Encrypted Virtualization(SEV)/Secure Memory Encryption (SME) [Adv18].

The Intel SGX trusted hardware platform consists of a set of microcode instructions, a dedicated trusted memory region in RAM and a set of hardware-infused keys. We begin with a description the Intel SGX programming interface, based on Intel's official documentation [Int19b]. We then briefly present the trusted hardware platform architecture, and specifically, how it provides the aforementioned guarantee to this interface, based on a very short summary of Costan and Devdas' detailed study [CD16] of the Intel SGX platform. We note that although we focus on the Intel SGX trusted hardware platform, our design can be implemented on other trusted hardware platforms, such as AMD's SEV/SME. To this end, we outline desirable properties of a trusted hardware platform for it to be used to implement our design. We conclude with a brief summary of known classes of attacks against trusted hardware platforms.

The Intel SGX programming interface involves partitioning native C/C++ applications into trusted and untrusted executables. Specifically, C-style functions in the application that may perform confidential operations or whose integrity needs to be guaranteed should be included in a trusted executable known as the *enclave*. The developer compiles and links the enclave executable in a special manner, which we detail below. The developer can deterministically compute a specific hash of the enclave executable, known as the *enclave measurement*. Other attributes of the enclave can also be specified in a configuration file. For instance, different versions of the same codebase or product have the same *Product ID* (ISVPRODID). Similarly, a *Security Version Number* (ISVSVN) attribute is used to demarcate versions that contain updates to address security vulnerabilities. The developer then signs over this enclave measurement and any attributes of the enclave specified in the enclave configuration file, to produce a signature token. The enclave executable and the signature token are then sent to an untrusted remote host. The developer can also easily compute a hash of the verification key corresponding to the signing key used to sign the enclave; this value is known as the *signer measurement*.

We briefly summarize relevant details from Costan and Devdas' study [CD16] in order to describe how Intel SGX securely loads enclaves and securely computes the above measurements.

A part of the DRAM, known as the Processor Reserved Memory (PRM), is made inaccessible to the OS, kernel, and peripherals, as it is passed through an encrypt-then-MAC function that uses a hardware-bound SGX key. The PRM hosts a small Enclave Page Cache (EPC) that stores enclave pages for multiple enclaves. The EPC contains a special page that the system software cannot map enclave pages into. This page stores metadata about enclaves, in an SGX Enclave Control Structure, or SECS. The trusted hardware platform loads the executable and the signature token into enclave pages in the EPC on the remote host. Enclave and signer measurements are initialized correctly by trusted microcode instructions within the SECS. In other words, as the untrusted OS cannot modify the SECS of the enclave, the platform provides the following guarantees to a developer in the secure remote computation context: unless the unmodified enclave is loaded, the trusted hardware platform on the remote machine will not report the correct enclave measurement. Secondly, unless the enclave's signature can be verified by the verification key in the signature certificate, the trusted hardware platform will not produce the correct signer measurement.

We can now proceed to understanding the programming interface in depth and the complex properties that can be obtained by trusting the SGX platform. Functions within an enclave are known as *ecalls*. Importantly, the enclave designer may not call any system calls within an ecall definition as these are handled by the untrusted kernel and are therefore not trusted to provide a correct return value or return arguments. However, the enclave logic may require calling system calls in order to perform I/O or network interactions. For this purpose an ecall may contain calls to functions defined outside the enclave, that is, within the untrusted executable. Functions defined within the untrusted executable that may be called into from within the enclave are known as *ocalls*.

A key security aspect relevant to this partitioning of code into the enclave and the untrusted application involves guarding pointers to arrays that are passed as input or output arguments to ecalls and ocalls. The programming interface requires the programmer to specify sizes of all input and output pointers. An SGX SDK `sgx_edger8r` program adds wrapper functions to ecalls and ocalls to ensure that:

1. Arrays referenced by pointers to untrusted memory that are passed as input arguments to ecalls or as output arguments to ocalls are copied to the enclave's EPC. This ensures that the enclave obtains one consistent copy of the variables and avoids time-of-check-to-time-of-use (TOCTTOU)[3] attacks.

[3]This class of attack stems from a race condition that allows an attacker to change data after it has been checked but before it has been used. In this context, if the array in untrusted memory were copied into the EPC each time it was referred to, or, even worse, if it were not copied at all and just referenced directly from untrusted memory, then the attacker could change the array between consecutive accesses.

2. As an untrusted application cannot observe enclave memory, arrays referenced by pointers to the enclave's EPC that are passed as output arguments to ecalls or as input arguments to ocalls are copied to untrusted memory.

The reader may refer to pages 7–18 of Intel's official documentation for SGX [Int19b] for further details about the programming model for Intel SGX and the pipeline for developing enclaves.

In addition to the above guarantee over the correctness of enclave and signer measurements, the Intel SGX trusted hardware platform provides the following guarantees, which we use to provide Mitigator's main guarantee.

1. **Sealing:** An enclave can save long-term secrets to disk such that the untrusted OS cannot obtain the plaintext secrets or modify them without being detected by the enclave, in a process known as *sealing*.

2. **Local attestation:** An enclave A wishes to attest to a target enclave B that it is indeed an SGX enclave that is specified by its enclave and signer measurements and is running on the same machine. This attestation is done in the presence of an untrusted OS that can observe and modify messages between enclaves.

3. **Remote attestation:** An enclave A wishes to attest to a developer that it is indeed an SGX enclave that is specified by its enclave and signer measurements and is running on a remote machine. Again, this attestation is done in the presence of an untrusted OS that can observe and modify messages between the developer's machine and the remote enclave.

In fact, Mitigator can be implemented on any trusted hardware platforms that provide features equivalent to the above three features. We proceed to discuss sealing; the reader may refer to Appendix A for details on how local and remote attestation are implemented within SGX. Sealing is done by performing an encrypt-and-MAC operation on the plaintext data, using a hardware-derived *sealing key*. Inputs to the sealing key can come from attributes of the enclave in its SECS page and a hardware-infused key. We briefly summarize design choices relevant to the sealing key [Int19b, pp. 28–32]; for further information on inputs to the sealing key, the reader may refer to §5.7.5 of Costan and Devadas's report [CD16]. The enclave designer may specify, as inputs to the sealing key, the enclave measurement or signer measurement or both. This choice controls the set of enclaves that can successfully decrypt or *unseal* this data. If the enclave measurement is specified as an input to the sealing key, then the sealed data can only be unsealed by other enclaves that have the same code. Such enclaves may be signed with a different key from the one used to sign the enclave that sealed the data. On the other hand, if the signer measurement is used as input to the sealing key, then the sealed data can only be unsealed

by an enclave that has been signed with the same key as that used to sign the enclave that sealed the data. Such an enclave may have a different enclave measurement; that is, different code. To allow an enclave that has been updated recently for a security vulnerability and thus has a different SVN, to unseal data that was sealed by it previously, the developer is allowed to specify the SVN of the enclave, as input to the sealing key. Any version of the original enclave whose SVN is greater than or equal to the one specified in the signature token can then unseal the data.

Finally, we observe that the Intel SGX platform has been repeatedly shown to be vulnerable to side-channel attacks. For instance, Lee et al. [LSG+17] show that an adversarial OS can infer enclave memory by inferring branches that have been executed in enclave code. Van Bulck et al. [BWK+17] show that an adversarial OS can infer accesses to enclave memory based on page table attributes and unprotected pages in its page tables. Van Bluck et al. [VMW+18] show that simply out-of-order execution of enclave instructions, along with side effects in caches, can be efficiently exploited by an adversarial OS to infer enclave memory. Defenses against side-channel attacks on the Intel SGX platform remain an active area of research amongst the security research community. For instance, Oleksenko et al. [OTK+18] propose restricting the adversarial OS from sharing CPU resources on cores that are used to run enclaves. To defend against a class of page-fault based and shared resource based attacks, Sasy et al. [SGF18] presents ZeroTrace, which is a memory controller to ensure that accesses to outside memory from within an enclave cannot be exploited to reveal data within the enclave.

Costan et al.'s Sanctum [CLD16] has been designed to minimize the attack surface and complexity inherent in the design of the Intel SGX platform and to specifically protect against cache and page-table based side channel attacks. Subramanyan et al. [SSL+17] present a formal abstraction of trusted hardware platforms and formally verify it to provide the core confidentiality, integrity properties and a property to ensure secure measurement of enclaves. The Keystone enclave project [LKS+19] is aimed at making open-source enclave designs that are resistant to physical and software-based side-channel attacks and are formally verified. Our threat model would be strengthened from research in defending enclaves against side-channel attacks. Finally, we remark that our design is general and can be implemented on other trusted hardware platforms in the future. We use the Intel SGX platform for our proof-of-concept implementation.

2.3.1 Supporting complex applications in Intel SGX enclaves

Several tools have been proposed to support running complex applications within Intel SGX enclaves. Arnautov et al.'s Scone [ATG+16], Tsai et al.'s Graphene-SGX [TPV17], Shinde et al.'s Panoply [SLTS17] and Lind et al.'s Glamdring [LPM+17] are all intended to run complex applications, in different frameworks in Intel SGX enclaves.

Supporting complex applications within Intel SGX enclaves, while continuing to provide guarantees of integrity of the code and confidentiality of the program state comes with two major challenges: first, placing a large application into an enclave directly will drastically increase the trusted computing base of the enclave, as a developer on a remote machine within the secure remote computation context now has to trust that all code within an enclave is free of vulnerabilities that can be exploited by an attacker to subvert the above guarantees. Therefore, applications' source code should be partitioned into untrusted main application code and trusted enclave code. Scone, Graphene-SGX, Panoply and Glamdring provide different programming models, some of which support automatically partitioning an application into trusted and untrusted code. The second main challenge in supporting complex applications within Intel SGX enclaves is to support system calls without breaking the aforementioned confidentiality and integrity guarantees and to do so while minimizing the TCB. To implement a system call, an ocall needs to be made to exit the enclave and this ocall can then call the relevant system call within an implementation of a C library. Graphene-SGX, Scone, and Panoply differ in the specific system calls that they support, the size of the C libraries that they refer to for the system calls, if at all, and consequently, their trusted computing base (TCB). We proceed to discuss these systems in detail with respect to how they address each of these two main challenges.

Graphene-SGX has been designed to support running unmodified applications within Intel SGX enclaves and therefore, all Graphene-SGX enclaves effectively contain only one ecall into the starting point of the application. Consequently, sanitization routines to ensure input arguments to an arbitrary ecall, like that included by the Intel SGX `edger8r` tool, are not included within the Graphene-SGX setup. However, if developers decide to partition their application by hand, then they need to manually [Tsa18a] include or reimplement such routines. Graphene-SGX enclaves include all dynamic libraries that the application depends on. They also include a manifest file which contains enclave configuration and security-relevant options. Graphene-SGX essentially builds support for Intel SGX enclaves on top of the Graphene library OS, which implements the Linux API by including system call wrappers over system calls exposed by the standard C library (glibc). As it includes a modified version of the standard C library within each enclave, Graphene-SGX has the largest TCB as compared to Panoply and Scone. A Graphene-SGX enclave can be configured to mount file systems and to access files within the mounted filesystem. Authenticated files are supported on Graphene-SGX by requiring the enclave developer to compute a hash of the file safely on their machine, mark the file as a *trusted* file within the manifest, and include its hash within the manifest. Graphene-SGX's system call wrapper for opening a file loads a trusted file into the enclave memory, computes its hash and ensures that it is the same as that in the manifest. The manifest can also contain files that are marked as *allowed* files, implying that their integrity is not checked at runtime. Graphene-SGX includes a somewhat

secure[4] implementation for the fork and exec system calls and supports enclaves that launch a predetermined number of threads, as was the case with the first version of the Intel SGX SDK. We remark that shared memory[Tsa18b] is not supported in the current version. In its current version, Graphene-SGX also does not provide support for local or remote attestation [Tsa17] and sealing [Isa18] for enclaves.

Similar to Graphene-SGX, Scone is intended to run unmodified applications within an SGX enclave. The authors intend for it to be used in a microservice-based architecture [Sco19a], wherein a large application consists of partitioned, single-process components in the form of several smaller microservices that interact with each other. They do not include any automated partitioning tools within their system. Dynamic libraries that are classified as protected are integrity-checked at runtime, similar to trusted files in Graphene's manifest. They provide sealing-based file encryption [Sco19b] as well as a centralized service, known as Configuration and Authentication Service (CAS) [Sco19c], to conduct local attestation with microservice enclaves and thereafter, to grant these enclaves with certificates to authenticate themselves to a developer on a remote machine. Their current design does not support [Sco19d] the developer in conducting remote attestation with the CAS in order to authenticate it; instead, the developer authenticates it based on the X.509 certificate shown as a part of the TLS connection to it. In this case, the developer needs to trust that the CAS does not exfiltrate the TLS private key outside of its enclave. Like Graphene-SGX, Scone supports up to a predetermined number of threads, but also efficiently multiplexes application-level threads such that they incur minimal latency while waiting for system call returns. Scone does not support the fork-related system calls.

Glamdring and Panoply both work within the model of the native Intel SGX platform with an untrusted application and trusted enclaves. Shinde et al. and Lind et al. [LPM+17] propose somewhat different automated ways to partition a large application's codebase using developers' annotations and static source code analysis to identify relevant functions and variables that should be put within the enclave. Shinde et al.'s partitioned enclaves are known as *microns*. Microns can be configured to share memory amongst each other and can be created on demand in order to support a dynamic number of enclave threads, even though the native Intel SGX SDK only supports a static number of threads. Shinde et al. do not include the C library within their TCB at all, as they intercept calls to the standard C library. They do include support for securely forking an enclave and as they work within the native SGX platform, the Intel SGX SDK libraries for local and remote attestation and sealing can be used by microns to implement these SGX-specific functionalities. As illustrated in their case study for porting the OpenSSL library to Panoply, parts of dynamic libraries can be run as microns, and the invoking application code can be modified

[4]The current implementation[Kuv18] of the fork system call transfers the image of the parent process to another enclave over an insecure channel and later authenticates the enclave.

Table 2.1: Comparison of platforms that support running complex applications within Intel SGX: - indicates no support, ◐ indicates partial support, and ● indicates full support.

Feature	Scone	Graphene-SGX	Panoply
Unmodified applications	◐	●	-
Fork	-	◐	●
Threads	●	◐	●
Local attestation	●	-	●
Remote attestation	-	-	●
Sealing	●	-	●

slightly to use Panoply's API to securely obtain the results of running functions within the library micron.

Importantly, support for running system calls as ocalls outside an enclave opens up the possibility that a malicious OS could tweak the return values of system calls to any other values, which can in turn lead to arbitrary code execution[5] Checkoway and Shacham [CS13] discovered this general class of attacks; they are known as *Iago* attacks. As Checkoway and Shacham illustrate, an adversarial OS could pass a pointer to a stack location instead of a heap location as a return value to a malloc call, resulting in arbitrary code execution. A defence against Iago attacks would require checking the returned values to ensure that a malicious return value is prevented from adversarially manipulating the control flow of the application. In Glamdring, the authors simply check the return value of the system calls against statically inferred return values. This does not protect against memory system call-based Iago attacks. Graphene-SGX, Panoply, and Scone include shields for system calls that return static as well as dynamic values, but do so for different fractions of the exposed system call interface. We summarize all design and security features that we have discussed so far for Scone, Graphene-SGX, and Panoply in the table below.

As we can see from Table 2.1, Panoply supports all of the relevant features, other than running applications without requiring any modifications. Additionally, as we remarked earlier, it has the smallest trusted computing base (TCB). We chose to use Graphene-SGX for Mitigator as we wanted to run multi-process unmodified applications within SGX. As we discuss in Chapter 5, we found that adapting Intel SGX SDK's libraries for local attestation and sealing was relatively feasible. Moreover, avenues remain open for adapting Panoply, Scone, and other platforms with lower TCB and/or more support for general-purpose applications with threads or forks, to work in future versions of Mitigator's implementation.

[5]It can also, of course, not respond to a system call and cause a denial-of-service on the running code. As mentioned in Section 4.1, providing availability is outside Mitigator's threat model.

2.4 Secure browser-to-user interfaces

Mitigator encrypts plaintext form field values to a verified server-side enclave. Other privacy-preserving browser extension systems also encrypt form field values to minimize the amount of plaintext data collection by websites, such that only other trusted or verified code can decrypt them. He et al. [HAJ+14] proposed a browser extension, named ShadowCrypt, to provide encrypted textual inputs to client web applications, effectively minimizing the amount of plaintext data that can be used by these applications to profile the user or to send to unauthorized third parties. Krawcieka et al. [KKP+18] propose SafeKeeper, which encrypts users' passwords to a trusted enclave on the server side. Users of such systems should be able to securely provide input to these fields in the face of untrusted scripts on the webpage or in malicious browser extensions. First of all, such untrusted scripts should not obtain the plaintext form field data. Secondly, in case the system is under attack, the attack should be detected and the webpage should not be changed to appear as if it was safe to enter form field data into. This avoids users from being misled into a false sense of security. We proceed with an analysis of the user interface designs proposed by these two systems, their drawbacks as presented in recent literature, and finally, the state-of-the-art approaches to overcome these drawbacks.

In both Krawcieka et al. and He et al.'s designs, symmetric keys used for encryption were stored within the extension and as browsers sandbox extensions and webpages separately, these keys could not be accessed by dynamic Javascript code on the website. He et al. used a new W3C standard was designed at that time, known as Shadow DOM, to enforce a boundary between users' plaintext data and the corresponding ciphertext, such that the former could only be accessed by the extension or, alternatively, that webpage scripts could only access ciphertext. ShadowCrypt's design involved making changes to the webpage for two purposes: to capture users' plaintext data securely and to provide feedback back to the user about this through changes in the website's appearance. Freyberger et al. [FHA+18] presented an attack on ShadowCrypt which demonstrated that changing the webpage for these purposes was insecure, in that untrusted scripts on the webpage could capture users' plaintext data and also mimic changes in the appearance of the webpage to mislead users into believing that they were providing inputs to the genuine ShadowCrypt extension. Their attack worked even in case the Shadow DOM boundary worked as intended. We proceed to discuss He et al.'s ShadowCrypt design features; that is, the Shadow DOM and the user interface, and Freyberger et al.'s attack.

A Shadow DOM is used to encapsulate the Document Object Model of a given element's subtree, in the form of a Shadow DOM tree. The Shadow DOM tree is attached to the given element, which is now known as a Shadow host. We remark that the browser renders the inner, encapsulated shadow DOM element in place of the shadow host. Secondly, a shadow DOM tree object can be prevented from being accessed by the shadow host, by setting relevant properties

at the time of its construction. He et al. use these two features of Shadow DOMs as follows: they detect text input elements on a webpage form and attach a Shadow DOM to each such element, thereby making the element a shadow host. Due to the first feature, users will see the shadow DOM tree object. Users are expected to ensure that they see a genuine shadow DOM tree field, based on its appearance, such as the border colour, highlight, small icons next to it, and if so, they are to enter plaintext data into it. Using the second feature, the plaintext form field data is made inaccessible to webpage scripts; the latter only obtain ciphertext data.

One of the aspects that Freyberger et al.'s attack relied on was the lack of complete mediation of all text input fields, which Freyberger et al. agree is difficult to provide. In particular, this allowed an untrusted webpage script to place a transparent text input field, which would not be detected by ShadowCrypt, *over* the Shadow DOM tree field in order to steal the users' data. Secondly, any changes in the appearance of the shadow DOM tree field can be replicated or imitated over the untrusted field. It is evident from Freyberger et al.'s user study on the stealthiness of this attack, that expecting a user to confirm that the browser extension functions correctly, through aforementioned in-page visual indicators, is not viable.

It is evident that a design that allows untrusted scripts on the original webpage to somehow overlay form fields over the intended ones in order to capture plaintext data and mimic their behaviour is potentially suseptible to Freyberger et al.'s attack. For instance, Freyberger et al. discuss utilizing sandboxed iframes[6] to present form fields to capture users' plaintext data, and remark that untrusted scripts on the webpage can spawn new iframes that can transparently overlay on top of the existing iframes and thereby steal users' data.

Krawcieka et al. propose a user interface design that is similar to He et al.'s in that it involves overlaying a form field over the original field to capture users' plaintext and as they rightly observe, is susceptible to an attack like Freyberger et al.'s. They also present another design to overcome this drawback, that requires the user to click on the browser extension's icon, which then results in opening up a pop-up browser extension window, where the user can enter the plaintext form field data. With the user-initiated interaction with the browser extension icon, an untrusted webpage script or an untrusted extension cannot observe whether the user has clicked on the SafeKeeper icon, or, observe or modify the plaintext form fields that the browser extension scripts on the pop-up receive. Most importantly, the webpage script cannot overlay a field over the browser extension pop-up, therefore rendering Freyberger et al.'s attack, in its current form, useless against their design. However, in this design, users would need to be trained to only enter plaintext data in SafeKeeper's browser extension pop-up and this pop-up would not have

[6]Iframes are HTML tags that open another window within the current window, in order to load content from another website. Using sandboxing, a new iframe can be loaded with the same URL as that of the original webpage and yet made to appear as if it came from a different origin.

the same look and feel as the original webpage. We use Krawcieka et al.'s design within the Mitigator implementation, as we discuss in Chapter 5.

2.5 Summary

In this section, we have discussed existing natural language processing-based tools for privacy policies and shown how they can be used within Mitigator to obtain a privacy policy model. We have also examined how source code analysis tools have been used in conjunction with the above tools to identify privacy policy violations on smartphone apps and can similarly be used to detect privacy violations for websites. Our discussion of trusted hardware platforms in general and the Intel SGX platform in particular has laid down concepts necessary to understand other systems aimed at addressing similar problems as ours, as well as our design. We conclude with observations from recent literature on implementing secure browser-to-user interfaces, which we apply later on in Mitigator.

Chapter 3

Related work

In this chapter, we discuss several systems that have been proposed to assess whether companies' source code is compliant to its internal standards or to external standards, such as legislation, privacy policies, or user-specified settings. We begin with an analysis of trust-based compliance, that is, trusting that an organization does what it claims to do by having a trusted third party verify its data handling practices in Section 3.1. We proceed to describe the P3P machine-readable privacy policy model and analyze its drawbacks, which emphasize the need for systems that ensure compliance of server-side code with the advertised privacy policy or privacy policy model. We then discuss systems that organizations may deploy to ensure compliance of applications running on its machines to internal or external standards. We finally describe systems that perform privacy-relevant operations, such as the aforementioned compliance check, on trusted hardware platforms, and use architectures that provide feedback to users of successful execution of program(s) that perform these operations. We conclude with thorough comparisons of these systems to Mitigator.

3.1 Trust-based compliance — internet seals

Various organizations grant verifiable icons known as *seals* to websites that fulfill their security or privacy standards, in order for the latter to assure website users of the security of their data or of the website providers' data handling practices. We start with a brief description of the current systems used by one of the most prominent internet-seal granting companies and then proceed to summarize shortcomings of seals, going by its historical failings, economic analyses by other experts, comparison of its seals with other representations derived from the natural language privacy policy, and importantly, its (mis)interpretations by users.

TrustArc, previously known as TRUSTe, provides a seal known as Enterprise Privacy Certification [Tru19b] to enterprises through a three-stage assessment, certification, and monitoring procedure. The current assessment procedure engages *human* experts, rather than any automation, to help an existing team identify the flow of users' information. Secondly, a program [Tru19a] aimed to guarantee that a website is compliant with privacy regulations, only *monitors* the customers' website through scans for trackers, cookies, and so on. In other words, it does not mandate analysis of the source code before issuing the seal for this program.

Historically, internet-seal issuing businesses have often granted seals without performing stringent checks on organizations. For instance, TrustArc (under its previous TRUSTe name) gave such a *privacy* seal to the Choicepoint data broker service, which trades personally identifiable information of thousands of people and organizations with businesses and law enforcement agencies. Choicepoint then suffered a data breach [McW02], when an internal database was released online without any authentication. Dangerously, TrustArc has relied on its customers' claims of compliance to their standards, whilst issuing seals, and thereby to relevant legislation. Consider the case of websites that are marketed towards children: these should not contain any online trackers or perform any behavioural advertising as these actions would be in violation of the Children's Online Privacy Protection Act (COPPA) of the United States. TrustArc's then informal process for its Children Privacy Program included ensuring that both of these components were absent from its customers' websites. However, its scans were malformed and did not check most or all webpages of its customers organizations and additionally, in some instances, TrustArc failed to make results of such checks available to its customer enterprises [Lew17]. Such a history of internet-seal issuing businesses illustrates that seals issued by such businesses may not be trustworthy in contemporary times.

In fact, Greenstadt and Smith [GS05] posit that as internet seal-issuing organizations are for-profit organizations that charge fees for granting seals to organizations, seals suffer from the economic phenomenon of capture. They explain that such a business is *disincentivized* to place more checks on an online business' data practices in order to obtain a seal, as existing online businesses would refuse to continue their business contract with the seal business, thereby reducing the latter's customer base. This has been recognized by Greenstadt and Smith as a disincentive for seal-issuing organizations from increasing the stringency of its standards. Harkous et al. [HFL+18] design privacy icons by extracting and interpreting privacy-relevant natural language clauses from the privacy policy and compare these with similar icons assigned by TRUSTe and Disconnect [Dis14] for the same websites around the same time. Both sets of icons contained one of three colours: red, yellow, or green. The results of their comparative study can be seen as substantiating Greenstadt and Smith's hypothesis: they found that the distribution of icon colours for their icons match those assigned by TRUSTe only when they assigned safer (green) labels for very permissive interpretations of the privacy policy clauses. Therefore, such seal-granting

businesses can be considered to be too lax in the standards that they expect their customer businesses to follow and as a consequence of this, seals may easily mislead users into a false sense of security.

In particular, not only have seal-granting businesses been found to follow lax policies in granting seals, users have been found to have misconceptions about the interpretations of seals. Kirlappos et al. [KSH12] conducted a study wherein participants were to browse a small set of shopping websites, which were split into two sets containing original websites and modified ones with additional seal images, for a particular item in a short time duration. They were to then assign ratings to these websites, corresponding to how much they trusted the websites to buy the item. The authors found that about a third did not notice seals on any of the websites and out of those that did, none verified their integrity. They found a positive, significant correlation overall between ratings for a website with a seal, versus for the website without the seal. Following the study, when participants' attention was drawn to the seals and when they were asked about their interpretations of the seals, about 12% of participants described seals as indicating that a website was legitimate, as they had seen other websites with such seals. The authors highlighted that "spillover" of trust leaves users vulnerable to mimicry attacks, wherein untrustworthy websites simply place such seals to trick users into believing that they are trustworthy. Therefore, it is evident that users may hardly notice seals and alternatively, they may consider a website as being trustworthy as it contained a seal that was also present on another, possibly more popular website (spillover of trust).

Mitigator enables providing users a guarantee based on the *source code* that operates on users' data, thereby bypassing the economic problem of capture faced by for-profit seal organizations. In general, any open-sourced program that can check the source code of the website, from a model derived from its privacy policy, can be used in Mitigator. In other words, Mitigator derives a signal of compliance that is only influenced from the source code of websites.

Secondly, Mitigator does not indicate this signal of compliance by embedding it within the webpage itself and it thereby avoids the simple mimicry attack which involves placing a Mitigator "seal" within a non-compliant webpage. Instead, Mitigator indicates this signal through a browser extension that we specifically design — based on current state-of-the-art research in secure user-to-browser interfaces — to not be mimicked even by other untrusted scripts on the webpage, let alone by other websites. In other words, if users see a prominent website as being supported by Mitigator and another less prominent website as also being supported by Mitigator, then the latter website must, indeed, be supported by Mitigator as it cannot have forged Mitigator's signal of compliance from the former website.

3.2 P3P: Machine-readable privacy policy models and the need for ensuring compliance

P3P [CLM+02] was deployed to be a format for machine-readable privacy policies that were to be processed by users' browsers. Users were to configure their own P3P policy within their browser and the browser would ensure that a website that a user visited had a privacy policy that was at least as restrictive as the user's. The browser that deployed P3P would block the website from setting cookies on the user's machine if its privacy policy was not as restrictive as the user's. Since P3P was developed, much research has gone into alternate forms of representation of privacy policies: for instance, Kelley et al.'s [KBCR09] nutrition labels, and various proposals for privacy icons, such as Harkous et al.'s [HFL+18] icons based on Polisis and Raskin et al.'s proposal for privacy icons for Mozilla [Ras10].

P3P has had a low adoption rate among websites at the time of its deployment and as Cranor found, it suffered from several deployment-relevant setbacks [Cra12, pp. 296–299]. For instance, the Internet Explorer browser did not check the syntactical correctness of P3P policies and simply disallowed policies based on certain terms in its blacklist. To prevent cookie blocking, website owners would copy syntactically incorrect P3P policies from other websites and blogposts in order to force P3P user agents in browsers to accept their policies. As Cannor concluded and as noted in the W3C standard for P3P, it was impossible to enforce that websites' P3P policies are representative of the websites' actual practices. She notes this as a reason why P3P's "privacy-by-policy" paradigm, which required website owners' diligence in posting accurate P3P policies, had to be complemented with a "privacy-by-architecture" paradigm that ensured only anonymized data would reach website servers or that most users' data would be processed locally on their own machines.

For instance, He et al.'s [HAJ+14] Shadowcrypt, which we discussed in Section 2.4, falls neatly in Cranor's classification as a "privacy-by-architecture" approach. It is based on the observation that web applications that provide text-editing and sharing services such as calendar tools, document and spreadsheet-sharing services, etc., do not need to know the data that users enter into text fields. Therefore, in their browser extension, they simply encrypt plaintext data entered by users into text fields to the extension itself. While this approach may minimize the amount of plaintext data available to such websites, it cannot be generalized to websites that operate on users' data, such as shopping or search websites. Mitigator supports such websites *as well*, by ensuring that the operations that their source code performs on plaintext data are compliant with a privacy policy model, before handing over this plaintext data to the source code.

It can be said, therefore, that Mitigator represents a design point in the middle of Cranor's two paradigms: website companies can, and should, work towards minimizing data collection. However, Mitigator provides guarantees *even if* the website provider collects and processes users' data on their server-side machines. Even though implementing privacy-enhancing tools within their website back end is complementary to Mitigator, Mitigator supports demonstrating to a *user* that the company is provably implementing such tools, as long as their correct functionality can be checked statically. Therefore, even though both P3P and Mitigator give users control over whether to send their personal data to a website, P3P bases the decision on whether the website's claimed privacy policy matches the user's preference, whereas Mitigator bases the decision on whether the website's backend source code matches its claimed privacy policy.

3.3 Systems for internal compliance

The Enterprise Policy Access Language (EPAL) [AHK+03] was designed to ensure compliance of programs that processed users' personal data within an organization with *internal* policies that were not known to users. They were also designed to ensure compliance when the data was transferred to third parties. We discuss two systems that have been developed since the proposal of EPAL to ensure compliance of programs running within an organization's servers to either a privacy policy model [SGD+14] or user-specified privacy-relevant settings [EMV+16].

Sen et al. [SGD+14] present Legalease, a first-order representation of privacy policies. Predicates in Legalease are over lattice encodings of user data types and operations, such as usage for a given purpose or sharing. Legalease can be used within Mitigator to encode textual privacy policies to a model. The authors follow a partially manual process to annotate data items with their types and scripts with the operations that they perform on users' data. This partially manual annotation then enables them to instrument an existing big data processing system for the search engine Bing, with an information flow tracking tool named Grok. Grok outputs the results of its data flow analysis across the entire distributed system to a policy checker. The policy checker then compares this output against the Legalease encoding of the privacy policy to report scripts that violate the privacy policy. Mitigator and Grok are similar in that they intend to achieve compliance of source code with a privacy policy model and to identify code that does not comply with the privacy policy. Elnikety et al. present Thoth [EMV+16], a reference monitor to mediate all system calls for files, that enables implementing access control policies derived from user-specified settings. Giffin et al. present Hails [GLS+12], a web framework that extends the Model-View-Controller paradigm with mandatory access control, implemented using type constraints native to the functional programming language Haskell. Their framework allows implementing privacy policy constraints over Model-View-Controller web framework programs

in the form of access control policies. Thoth and Hails are similar in that they remove the programmers' burden of maintaining access control-like policies scattered throughout different programming languages that implement the website (Thoth) or various source code units of one language (Hails) and instead group all such policies into a separate, configurable unit.

In contrast with both Grok and Thoth, Mitigator does not involve any runtime checks on users' data. Similar to Mitigator, Hails effectively implements static checks on the program. Importantly, in contrast with Grok, Thoth and Hails, Mitigator has a stricter threat model, namely that of an adversarial OS and therefore, a smaller trusted computing base (TCB). Thoth, for instance, requires trusting the reference monitor, a kernel module, and the rest of the Linux OS. Even though Hails checks the website's source code files against the access control policies, a successful check is only meaningful if the website source code is not somehow replaced with malicious source code at runtime, thereby effectively requiring users to trust the programming language interpreter and the entire server-side OS. Mitigator anchors trust in a subset of the hardware and thus does not require trusting the OS. Finally, as it stands, users of a Grok/Legalease or Thoth system do not know of its existence as these systems do not provide users feedback, whereas Mitigator is designed such that users of a website that has deployed it can easily know of its existence. Furthermore, we design Mitigator to also prove to users, basing trust on the smaller TCB, that it is functioning correctly. Thus, the equivalent of the above file replacement attack on a Hails-supporting system will be detected in Mitigator. We proceed to discuss other related privacy-enhancing systems built with trusted hardware.

3.4 Trusted hardware platform-based compliance systems

Trusted hardware platforms, including Intel SGX, have traditionally been designed to implement Digital Rights Management (DRM) tools that enforce restrictions on what consumers of digital media can watch. Korba and Kenny [KK03] proposed appropriating a DRM system on the server side to implement users' preferences over the usage and transmission of their data through a Privacy Rights Management (PRM) system. These two tasks are performed within a *PRM server*, with the user being able to configure this component through a web interface.

More concretely, Maniatis et al. [MAF+11] propose Secure Data Capsules: data *capsules* that contain data items along with a provenance log of operations conducted on them and a set of policies that determine who can read, update, or declassify the data items. Maniatis et al. conceptualize a Secure Execution Environment (SEE) unit on a trusted hardware platform that consists of unmodified, legacy applications and a reference monitor to monitor system calls made by the application to access or modify data. Users authenticate themselves to the SEE through an *authentication manager* that is also run within the trusted hardware platform. The

SEE reference monitor communicates securely with a *capsule manager*, based on attestation, to request data items on behalf of the authenticated user. We have seen that Elnikety et al.'s Thoth is a reference monitor-based information flow tracking system. Maniatis et al.'s Secure Data Capsules envisions such a reference monitor-based information flow tracking system within a trusted hardware platform. Mitigator differs from Secure Data Capsules in that it supports privacy-policy compliant processing of users' data in the context of websites, that is, when the user does not directly interact with the trusted hardware platform and nevertheless, provides users proof of only allowing privacy-policy compliant applications to access their data. Secondly, rather than denying access to users' data to an application at runtime, we check the application's source code for, and enforce compliance to, the privacy policy beforehand, through static source code analysis.

Kannan et al. [KMC11] present Secure Data Preservers to allow users to process their data with their own privacy-preserving programs, known as *preservers* on a virtual machine (VM) or on a trusted hardware platform on server-side machines. They present several small programs, among which one anonymizes users' data by pooling it with data in other preservers. A user can dictate a *hosting* policy that is required to hold true on a server side machine for it to be able to instantiate and run the user's preserver. In particular, the user communicates with a *base layer* that is situated on a VM or a trusted hardware platform. If the user requires the server-side machine to have a trusted hardware platform in its *hosting policy* and it does, then the base layer on that machine performs remote attestation with the user and consequently, the user hands over its preserver over the established secure channel. Interestingly, Kannan et al. also support applying preservers as users' data crosses organizational boundaries, as follows: the base layer performs a two-sided remote attestation with a similar base layer that satisfies the user's hosting policy and transfers the preserver to it. The base layer is trusted to also implement user-defined *invoking* policies, that determine who can invoke the preservers' interfaces, thereby effectively implementing an access control layer over them.

As Kannan et al.'s preservers are designed to be lightweight, they contribute little to the TCB in comparison to the base layer. However, as a preserver is created for *each* user, this design requires significant memory: for instance, following extensive deduplication of VM pages across preserver VMs for users that share preservers, they found an example preserver to take about 10 MB. This is a significant barrier for preserver hosting policies that require trusted hardware platforms like Intel SGX: as the EPC is very limited in size (90 MB), only a few preservers can be run on the same machine. Moreover, in the absence of further experiments, the runtime latency costs for instantiating and running a preserver on a trusted hardware platform as per the above process, which involves the base layer, are unknown.

Most closely related to Mitigator in the context of privacy-preserving browser extensions is Krawcieka et al.'s [KKP+18] SafeKeeper. Like Mitigator, through the use of an authenticated

trusted hardware enclave on the server side, SafeKeeper provides evidence of its integrity to a remote user. The authors cater to the problem of protecting passwords against compromised servers, which may be susceptible to offline and online guessing attacks, as well as phishing attacks. They run an enclave that establishes a secure channel with the SafeKeeper browser extension through remote attestation. The browser extension may then verify SafeKeeper's server side enclave's enclave and signer measurements from the remote attestation quote and thus assure a user that a valid SafeKeeper enclave is running on the server side.

A SafeKeeper enclave accepts an encrypted password and a randomly generated salt as inputs to an ecall that first decrypts the password with the remote attestation-based session key with the client. It then computes a CMAC [Dwo16] over the plaintext password and the salt, using its key. If the user wishes to set a password, then the CMAC output is stored in the password database. Otherwise, it is compared to the CMAC for the corresponding salt in the database. Therefore, SafeKeeper stores the CMAC key using a trusted hardware platform such that a rogue or a compromised server cannot guess the passwords, while simultaneously assuring the user of the existence of a valid SafeKeeper enclave through remote attestation. We observe, however, that the passwords are modified in an irreversible manner, through the computation of the keyed CMAC and desirably so. Mitigator generalizes SafeKeeper in the sense that general operations on incoming encrypted data from clients are supported as long as these operations are compliant with the displayed privacy policy.

Birrell et al.'s work [BGR+18] in the context of enforcing use-privacy is similar to Mitigator in that they run the target application that processes users' data and a trusted program to enforce compliance of the target application with policies, within Intel SGX enclaves. Specifically, the authors work within a model where the trusted program safeguards access to a data store. The trusted program is a reference monitor that contains a predefined policy store, with a mapping of each target application enclave, as defined by its enclave and/or signer measurement, to its expected purpose of use of its inputs. Within the source monitoring model, the target enclave requests a remote monitor for a specific data item d, for a claimed purpose of use P along with a remote-attestation based quote q. The monitor first verifies the quote and then compares P and q with the values specified within the use-based policy for that data item d.

Although Birrell et al.'s model may seem similar to ours at the first glance, there are fundamental differences: first, we enable mediation of users' data as it is collected by a target website, rather than after it has been consolidated into a data store. Their paradigm of use-privacy only focuses on various uses of data, whereas we enable detecting privacy violations other than the secondary use violation. Second, as described in the context of Secure Data Capsules, our system's design does not involve performing runtime checks on and tracking of data that flows into

an application or an enclave: instead of maintaining a policy store[1] and a runtime monitor, we offload the task of ensuring compliance of the target application to another enclave on the remote machine. As long as static analysis based tools can detect all violations of the privacy policy model, we present an approach that only involves a one-time check with each update in the source code of the target application. To achieve compliance, developers need to state operations on users' data in a privacy policy that is accessible to users and indirectly, to minimize the amount of data collected in the first place. Finally, we support *users* in knowing that our system is correctly functioning on the target website. We do so by providing the client enough information to ensure that the users' data is handled by a target enclave that is compliant with the displayed privacy policy, as checked by a genuine, open-source program. On the other hand, Birrell et al. do not include any mechanisms to provide feedback of the existence of their systems to a remote user. We contrast elements of the design of their schemes against our design in Chapter 6.

3.5 Summary

In summary, we have described how trust-based compliance through internet seals are inadequate in giving users reliable signals of compliance of organizations' server-side applications with the privacy policy. Through our summary of P3P and its drawbacks, as reported by experts in related literature, we have established that machine-readable privacy policy models do not mandate website providers to ensure compliance of their source code with the advertised privacy policies. On the other hand, systems for ensuring compliance internally, within an organization, to privacy policy models or privacy-relevant settings do not provide a feedback signal to users of their existence or efficacy. Finally, we examine related work that anchor users' trust in hardware platforms to deploy privacy-preserving programs on their data on such platforms on the server side.

[1]The policy store mappings need to be updated with the expected measurements of a given target enclave following each update in the latter, resulting in a high maintenance cost.

Chapter 4

Model and design

Mitigator aims to provide a trustworthy guarantee to users that only website source code that complies with its privacy policy can obtain access to their form data. Our utilization of the Intel SGX trusted hardware module in Mitigator's server-side components is central in providing this guarantee. We begin this chapter with a bottom-up approach to motivate Mitigator's server-side and client-side architecture. We do so by beginning with a simple application of the guarantees provided by a trusted hardware platform to our scenario of ensuring and illustrating compliance of the website server code with its privacy policy.

We start with a program that given the privacy policy file and source code files for the target website as inputs, checks the source code files for compliance with the privacy policy text. Internally, this program may convert the privacy policy text into an appropriate model, using one of the methods outlined in Section 2.1. It may then provide this model as an input to a source code analysis routine, such as the ones discussed in Section 2.2, that would check that the source code files handle users' data as per this model.

We now switch to the context of trusted hardware platforms and we start with reiterating our trust assumptions. We assume confidentiality of the program state of a program running within a trusted hardware platform and integrity of the program. In particular, evidence of the integrity of the program is given through local or remote attestation. We assume that if the correct enclave and signer measurements are reported through local or remote attestation, then only the instructions of the intended enclave executable and no other instructions have been loaded into the enclave.

The usual secure remote computation use case for trusted hardware platforms involves the client as the enclave developer, who generates and signs over the enclave executable on a *trusted* machine. We use the trusted hardware platform for a different use case: we wish to sign an

enclave, say enclave B, from *within* another enclave, say enclave A. Specifically, enclave B is an enclave that runs the website server, which in turn runs some source code files and supplies a privacy policy file. Enclave A runs the verifier program on these source code files and the privacy policy model file. The verifier program is extended to produce a signature over the aforementioned files and the website server executable *only* if it finds them to be compliant. We assume that this verifier program is sound, but not complete. In other words, it will only produce a signature over the target enclave if it is compliant with the privacy policy. As the verifier program may be incomplete, it may wrongly deny signing valid target enclaves. We observe that this affects the usability of our tool by developers as they will need to simplify or annotate the target enclave source code or privacy policy model until the verifier program accepts the target enclave as valid with respect to the privacy policy model. We start with a strawman scheme in order to motivate our proposed design.

A strawman scheme: At this point, we have two programs running on the website provider's untrusted machines; we refer to enclave A as the verifier enclave[1] and enclave B as the target enclave, such that the verifier enclave has signed the target enclave. The client computes the enclave measurement of an open-source verifier enclave. It then performs remote attestation with the verifier enclave and ensures that its enclave measurement, as reported in the quote, is what it expects from the open-source version. In the secure channel established through remote attestation, the verifier enclave sends the signer measurement of a target enclave, that is, it encrypts this signer measurement to the client using the symmetric key established through remote attestation. A vulnerable or malicious program on the untrusted machine, or even the operating system, cannot learn the remote attestation-based key as it will be stored within the verifier enclave. Without this key, these parties cannot modify this message without being detected by the client.

Therefore, following this remote attestation with the verifier enclave, the client learns the expected signer measurement of a valid target enclave. The client then performs remote attestation with the target enclave and ensures that its signer measurement is the same as that sent by the verifier enclave. The client now sends the user's form field data to the target enclave over this channel. In other words, the client encrypts the form field data to the target enclave using the symmetric key established through remote attestation. Again, as the untrusted operating system on the remote machine cannot recover the symmetric key from within the target enclave, the user can be assured that only a valid target enclave will process their form field data. Moreover, based on the aforementioned soundness assumption, namely that the target enclave is assumed

[1]Even though we borrow terminology from the Intel SGX platform for programs running within a trusted hardware platform, our design remains general across trusted hardware platforms that provide the aforementioned guarantees.

to be compliant if the verifier enclave has signed it, the user can be assured that their data will be processed (by a target enclave) in a manner compliant with the privacy policy.

We remark that the client must perform remote attestations with both verifier and target enclaves before sending any form field data. If the client only performs attestation with the verifier enclave, then the following stealthy attack is evident: the untrusted OS may simply sign and run its own non-compliant website server enclave, which may perform privacy-invasive operations on the user's data, instead of the compliant target enclave that was signed by the verifier enclave.

Some inefficiencies: The above scheme is quite inefficient as it would require the client to perform two remote attestation protocols with two server-side enclaves, before submitting any form field data. We make the following observation that leads us to an improved design which only requires the client to perform one remote attestation per website. Through the second remote attestation, the client essentially authenticates the target enclave, based on its signer measurement that it obtains from the verifier enclave through the secure channel established from the first remote attestation with the latter. We can simply offload this task of authenticating the target enclave to another enclave on the website provider, which we refer to as the decryptor enclave. This leads us to our improved design.

Mitigator design: For both of the target and verifier enclaves, in case when the decryptor enclave is on the same machine as the given enclave, it performs local attestation with that enclave. Otherwise, it performs remote attestation with it. Guarantees provided by this design are not dependent on whether the decryptor enclave is on the same machine as the other two enclaves or not. For simplicity, we suppose that the verifier, target, and decryptor enclaves are all on the same machine. In this case, the decryptor enclave performs local attestation with the verifier enclave and obtains the expected signer measurement of the target enclave. It then performs local attestation with the target enclave, and checks that the reported signer measurement is the same as that sent by the verifier enclave. It follows that local or remote attestation results in a secure channel between the two attesting enclaves — in this case, in between the decryptor and the target enclave, as well as between the decryptor and the verifier enclave — but not between the client and the decryptor enclave. Therefore, it is necessary in this revised design that the client knows the expected enclave measurement of this decryptor enclave and performs remote attestation with the decryptor enclave to check that its enclave measurement is what the client expects it to be. Following the remote attestation, the client can use the shared secret established through it, to encrypt the users' form data to the decryptor enclave. As its name suggests, the decryptor enclave simply decrypts the data and re-encrypts it to the target enclave.

Unlike the simple protocol, in the above protocol, the client does not need to conduct remote attestation once before *each* time that the user visits a website and thus, we optimize it further, as follows. The client may perform remote attestation with the decryptor enclave behind each

Mitigator-supporting URL in an asynchronous manner, that is, independent of the time when the user accesses it. The decryptor enclave sends a long-term verification key through the remote attestation secure channel. The corresponding signing key may then be used to sign enclave-generated or attested secrets that the decryptor enclave then passes to the target enclave to send to the client with each HTTP response. In particular, the client needs to conduct this remote attestation only when the aforementioned verification key changes. Assuming that the plaintext key is not leaked outside of the enclave, this will happen only if the decryptor enclave is updated or if it is run on a different machine. Furthermore, users can trust a privacy advocate organization to perform the remote attestation, certify that they have verified this website's decryptor enclave's enclave measurement is as expected and publish its verification key.

In our design, the decryptor enclave generates a short-term public key and concatenates it with the verifier's enclave measurement and a hash of the privacy policy. It signs the concatenated value with the long-term signing key. Signing over the enclave measurement allows the client to decide if it wishes to communicate with a target enclave checked for compliance by the particular server-side verifier enclave. Similarly, signing over a hash of the privacy policy lets the client ensure that the privacy policy displayed on the website is indeed the one against which the verifier has checked the compliance of the target enclave's source code. Following a successful verification of the signature over these three signed values from the decryptor enclave, the user knows that the displayed privacy policy is the one against which a trusted verifier has checked the source code of the website for compliance.

We have introduced three enclaves that form Mitigator's server-side architecture: the *verifier* enclave, the *decryptor* enclave, and the *target* enclave. As mentioned previously, for simplicity, we assume that all of these enclaves are on the same machine. We have also referred to several roles of the client, namely obtaining a signature by the decryptor enclave over three values, verifying the signature and signed values, and using the key to encrypt content to the decryptor enclave. These tasks are perfomed by a *browser extension* on the user's machine. Before we go into details of these four components, it is important to elucidate the threat model that Mitigator is based on. We proceed to detail the threat model in Section 4.1. We describe our system design in detail in Section 4.2 and then in Section 4.3 we illustrate how our system defends against the adversary defined in the threat model. Finally, we discuss the repercussions of adopting alternative choices for certain elements of our design in light of how they affect the above analyses in Section 4.4.

4.1 Threat model

Mitigator does not require users to trust the server-side machines, other than a small trusted hardware module. In particular, employees of the website provider on the server-side machines may run privileged or unprivileged processes in an attempt to undermine the guarantees provided by Mitigator. We encapsulate this by treating the operating system itself as a privacy policy-violating adversary whose intent is to obtain the user's plaintext form field data and use it in violation of the privacy policy. The adversary may also attempt to mislead the user into believing that the verifier enclave checked the source code for compliance with a more restrictive privacy policy. If it succeeded in doing so, then the user may be deceived into a false sense of security about the website's data handling practices. We remark that we limit the adversary's intent to only performing attacks that result in the users' data being used for different purposes or disseminated to unintended recipients. An adversary who intends to spam the user with any material in violation of the privacy policy may be thwarted with a small change to the decryptor's design, as we describe later on.

Our privacy policy-violating OS can observe any unencrypted communication between processes or replay encrypted messages. As the TLS connection may reasonably be terminated outside the enclave,[2] the OS can modify any unencrypted HTML content as well as replay any encrypted content to the target enclave. It can also manipulate the file system and return values of system calls to whatever it desires. For instance, it can modify or delete files, or insert new files. Similarly, in performing a system call to write to a socket, it can modify values before writing them. The kernel can also manipulate its data structures to facilitate an attack that would undermine Mitigator's guarantees.

We consider an attack to be successful when a non-compliant website is falsely flagged by Mitigator as a compliant one. However, we allow compliant websites to be wrongly marked by Mitigator as non-compliant, without considering this as an attack on Mitigator. In other words, Mitigator is assumed to be sound for illustrating compliance of compliant websites, but it is not assumed to be complete. Finally, we restrict these attacks by an adversarial OS to exclude all side-channel based attacks: as we mentioned in Section 2.3, much recent literature surrounding trusted hardware is directed at preventing and mitigating these attacks. The privacy policy-violating OS may attempt to simply prevent the system from running, by performing denial-of-service (DoS) attacks on various enclaves or on communication between enclaves. As mentioned in Chapter 1, we assume that the website provider has incentives to deploy Mitigator such as ensuring that their website code complies with legislation or to reassure their users on the

[2] Although Aublin et al. [AKO+17] propose TaLoS to terminate TLS connections within the enclave, this requires the TLS private key be generated within the enclave and not exported elsewhere. It remains to be seen whether systems like TaLoS become widely deployed, considering this possible drawback.

company's transparency. We assume, therefore, that it is not their interests to perform such DoS attacks on their own deployment of Mitigator and that in case the attacks occur due to unforeseen circumstances, the organization will attempt to identify and stop them.

We also bound our adversary computationally; that is, we assume that the adversary is bounded by the hardness of cryptographic problems such as the discrete log problem. In assuming the existence of symmetric and asymmetric key cryptography, Mitigator's design inherits the hardness assumptions of the symmetric-key cipher, the signature scheme, and the Diffie-Hellman key exchange method chosen for implementation. Similarly, we assume collision resistance of standard cryptographic primitives. Trusted hardware modules similarly assume the hardness of certain cryptographic assumptions. The weakest assumption of all assumptions in Mitigator's implementation and the trusted hardware module, will determine the computational bound for our adversary.

Additionally, any tools on the client side, such as the browser extension, are assumed to be run correctly. In particular, we assume that the browser extension is being run without any manipulations of its source code in the browser. That is, an attacker cannot deceive a user by manipulating the user's machine into falsely displaying a non-compliant website as a Mitigator-approved website. We detail the trusted computing base (TCB) for Mitigator in the implementation chapter as it is dependent on the systems that we use to run the enclaves on the trusted hardware platform and the tool that we use within the verifier enclave to verify compliance. We now proceed to describe the functionalities of the four components of Mitigator.

4.2 System design

We start with briefly describing three chronological stages for Mitigator. The first two of these stages involve Mitigator components at the server side. Specifically, the first stage is to be run as the website's source code is being developed and in this stage the verifier enclave checks the given source code and privacy policy for compliance. In case of compliance, it outputs a signed target enclave. In case of non-compliance, the developers are expected to modify their source code and privacy policy and rerun the system until it passes as compliant. They are also expected to continuously rerun it as the source code is updated. The second stage occurs once the target enclave is ready to be deployed. In the second stage, the decryptor enclave conducts local attestation with both enclaves. It maintains the local attestation derived key for the channel with the target enclave for the third stage. The third stage refers to runtime interactions between the client, target, and decryptor enclaves. In particular, in the third stage, the client passes encrypted form field data to the target enclave, which in turn hands it over to the decryptor enclave over the aforementioned secure channel. The first, second, and third stages are named the verification,

the deployment, and the runtime stage respectively. We depict a high-level block diagram of our system in Figure 4.1. We now go on to detail each of these three stages below.

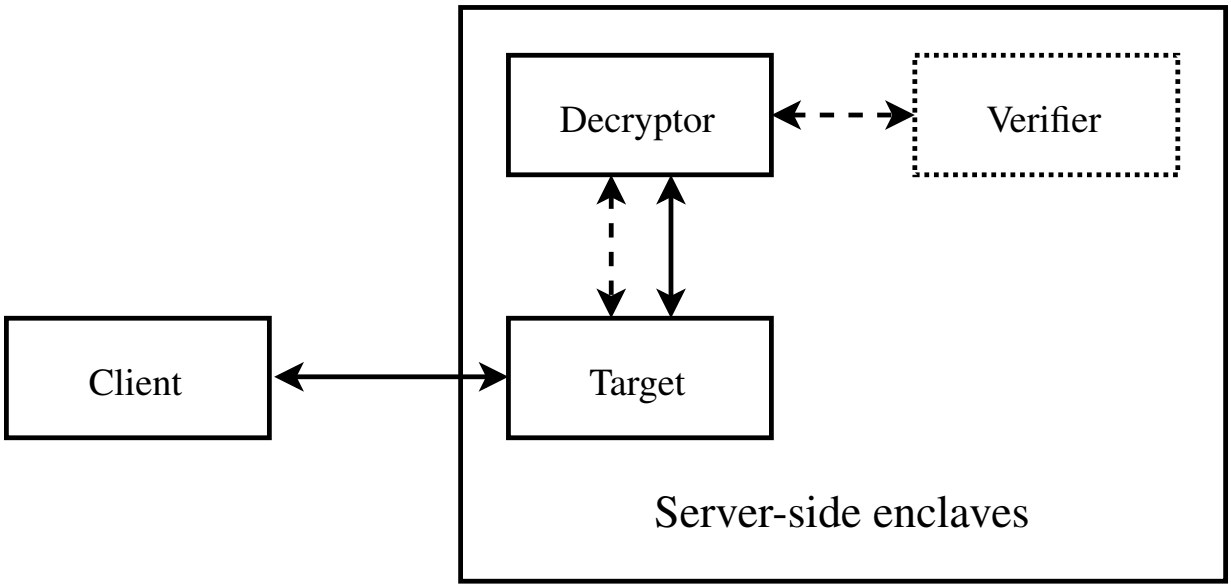

Figure 4.1: A high-level block diagram of our system. The component in the dotted box will be online in the verification stage. The dashed arrows indicate interactions occurring in the post-verification stage, which are detailed in Figure 4.3. The solid arrows indicate interactions occurring in the runtime stage, which are detailed in Figure 4.4.

4.2.1 Verification stage

As this stage occurs concurrently with the development of the target enclave source code and the privacy policy and involves the verifier enclave checking these, only the verifier enclave needs to be online during this stage. The verifier processes its input, namely, a list of source code files and the privacy policy. It passes these as inputs to the compliance-checking algorithm, which returns a binary value for compliance. Preferably, in case of non-compliance, this algorithm provides feedback for developers, including line numbers and other relevant information on non-complying source code. If the source code is indeed compliant, the verifier obtains its long-term keypair, as discussed below. It signs the target enclave with its long-term signing key, $\mathcal{SK}_{\mathcal{V}}$. We illustrate the verification stage in Figure 4.2.

The verifier enclave follows Algorithm 1 for obtaining a long-term signing-verification key-pair $(\mathcal{SK}_V, \mathcal{VK}_V)$. It first looks for a previously sealed keypair file on disk and attempts to unseal

it. We note that we use the verifier enclave's enclave measurement as one of the inputs to the sealing key. If the keypair can be found and unsealed successfully, then it proceeds to use the signing key $\mathcal{SK}_V$ later to sign over the target source code files. If a keypair is not found on disk or the unsealing is not successful, then it generates a new keypair $(\mathcal{SK}_V, \mathcal{VK}_V)$. The unsealing may not be successful if either the OS attempts to modify the keypair file or if the verifier enclave has been updated and so it no longer has the same enclave measurement required to correctly generate the sealing key. After generating a new keypair, the verifier enclave seals it to disk, using its enclave measurement.

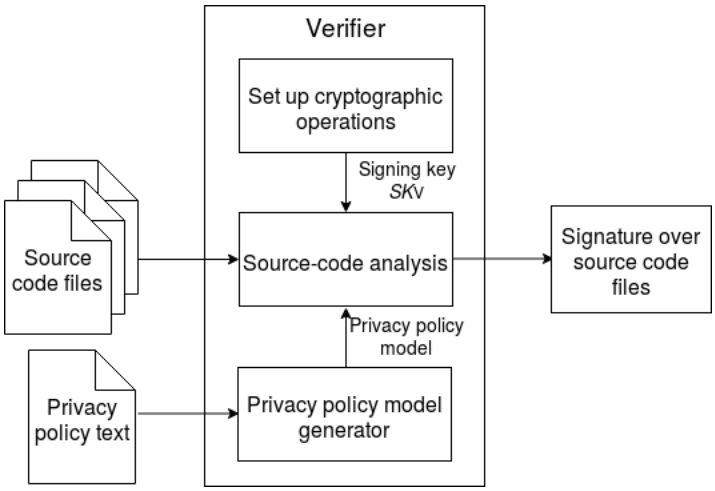

Figure 4.2: Mitigator verification stage — verifier block diagram

Algorithm 1 Algorithm to obtain a long-term keypair. The algorithms for the $ecall\_generate\_and\_seal\_keypair$ and $ecall\_unseal\_and\_set\_keypair$ functions are in Appendix B.

```
1: ▽ This is an untrusted function.
2: function SET_SIGNING_KEYPAIR(keypair_file_path)
3:     if file_exists(keypair_file_path) then
4:         sealed_keypair ← read_file_at(keypair_file_path)
5:         unseal_successful ← ecall_unseal_and_set_keypair(sealed_keypair)
6:         ▽ File has been modified or enclave has been updated.
7:         if unseal_successful == False then
8:             generate_sealed_keypair(keypair_file_path)
9:         end if
10:    else ▷ File does not exist or adversarial OS performs a DoS.
11:        generate_sealed_keypair(keypair_file_path)
12:    end if
13: end function
14: ▽ This is an untrusted function.
15: function GENERATE_SEALED_KEYPAIR(keypair_file_path)
16:    sealed_keypair ← ecall_generate_and_seal_keypair()
17:    write_to_file_path(keypair_file_path, sealed_keypair)
18: end function

19: global variables ▷ These variables are within the enclave.
20:    verification_key
21:    signing_key
22: end global variables
23: ▽ This ecall unseals the signing-verification keypair passed as the input argument. If it is
    successful in unsealing the keypair, it sets the signing_key, verification_key variables to
    the signing and verification key respectively and returns True. Otherwise, it returns False.
24: function ECALL_UNSEAL_AND_SET_KEYPAIR(sealed_keypair)
25: end function
26: ▽ This ecall generates a signing-verification keypair and sets the signing_key,
    verification_key variables to the signing and verification key respectively. It then returns
    the sealed keypair.
27: function ECALL_GENERATE_AND_SEAL_KEYPAIR()
28: end function
```

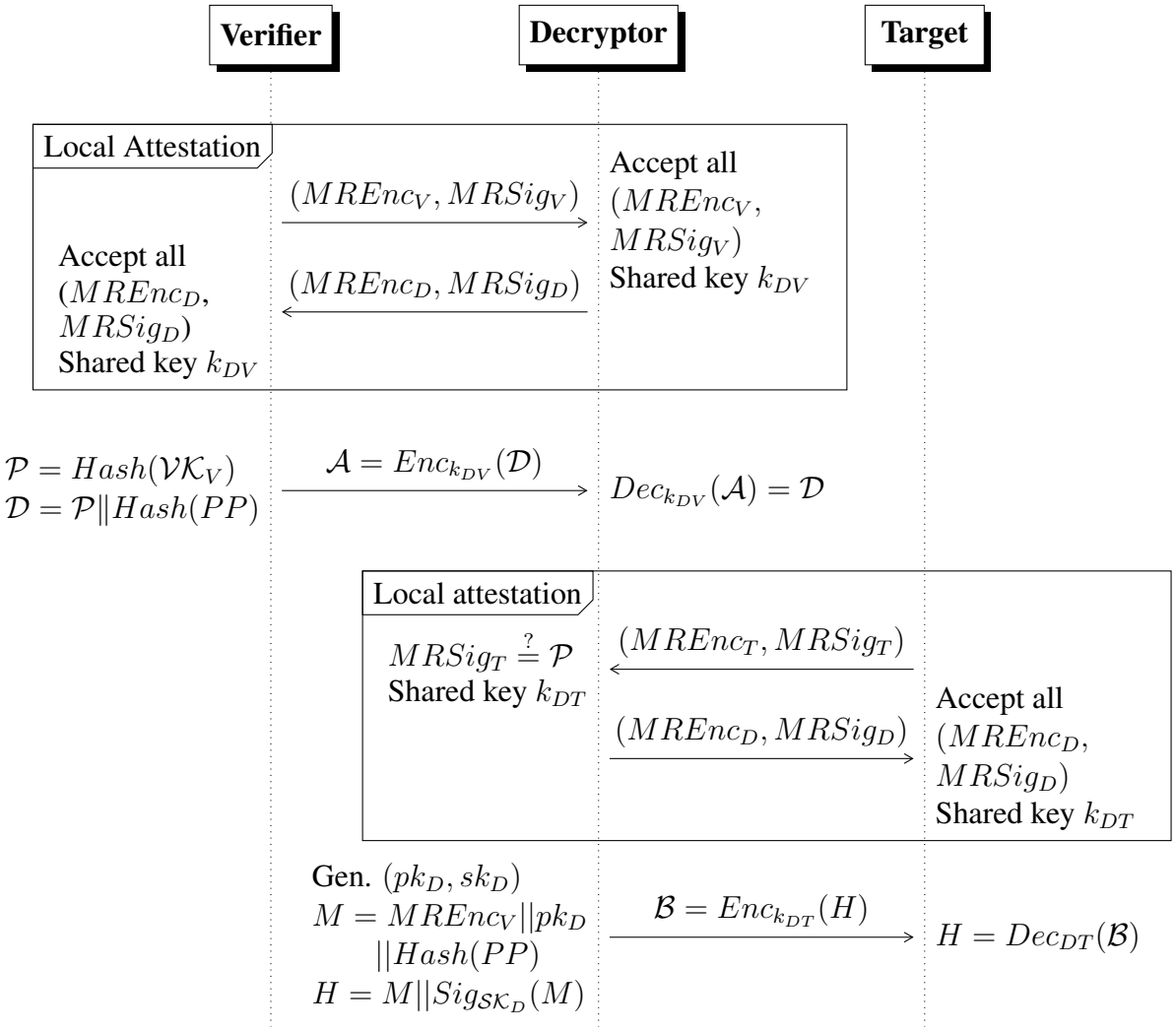

Figure 4.3: After the verifier enclave signs over a compliant target enclave with the keypair $(\mathcal{SK}_V, \mathcal{VK}_V)$, it initiates a local attestation request with the target enclave. The decryptor enclave then communicates the token H to the target enclave over the secure channel established through the attestation. Following this, the target enclave may conduct local attestation with the decryptor enclave. The decryptor enclave sends a token $\mathcal{B}$ to the target enclave. These steps occur before any client connects to the website and enable the target enclave to forward the token $\mathcal{B}$ to the client.

4.2.2 Deployment stage

In this stage, the verifier, decryptor, and target enclaves need to be online. The decryptor enclave follows the same algorithm as the verifier enclave in every invocation to obtain its long-term signing verification keypair. When the decryptor is run for the first time, it generates a long-term signing-verification keypair ($\mathcal{SK}_D, \mathcal{VK}_D$). The decryptor then seals this keypair to disk, using its enclave measurement as an input to the sealing key. It attempts to unseal this keypair for its subsequent invocations. Mitigator's browser extension obtains the verification keypairs of decryptors of various Mitigator-supporting websites. The decryptor enclave should have a remote attestation service open for clients to connect to it, so that clients can verify its enclave measurement through remote attestation and consequently learn its verification key $\mathcal{VK}_D$ over the secure channel. After setting up its long-term signing-verification keypair and the above service, the decryptor enclave starts listening for local attestation requests. A sequence diagram of messages sent and received in this stage is shown in Figure 4.3.

Verifier enclave to decryptor enclave

After signing the compliant target enclave, the verifier enclave conducts local attestation as an initiator with the decryptor enclave. The verifier enclave thereby forms a secure channel with the decryptor enclave, through a symmetric encryption key k_{DV} derived from the local attestation shared secret. Over this channel, it shares the expected signer measurement over the source code files, that is, a hash of the verification key, $Hash(\mathcal{VK}_V)$. Additionally, the verifier enclave also signs and sends a hash of the privacy policy, $Hash(PP)$. This is because as the adversarial OS can manipulate HTML content that is not encrypted to one of the three enclaves, or it may attempt to replace the privacy policy HTML file against which the verifier checked the source code with a more restrictive privacy policy file. As we mentioned in Section 4.1, we cannot allow the adversary to mislead the user into believing that Mitigator checked the website source code against a stricter privacy policy. Signing a hash of the privacy policy allows the client to detect the above attack.

Suppose that as per the local attestation report sent by the verifier enclave to the decryptor enclave, it has an enclave measurement $MREnc_V$ and signer measurement $MRSig_V$. The decryptor enclave does not compare $MREnc_V$ or $MRSig_V$ to any hard-coded values, but simply stores $MREnc_V$. After the first successful local attestation, the decryptor expects the initiating enclave to respond with two hashes encrypted to k_{DV}. It treats the first hash as a signer measurement of a compliant target enclave, that is, as $Hash(\mathcal{VK}_V)$. The second hash, $Hash(PP)$, is stored along with the verifier's enclave measurement $MREnc_V$ and is treated as a hash of the privacy policy against which the target enclave was checked for compliance. The decryptor

enclave then terminates the secure channel with the verifier enclave and waits for a second local attestation request.

Target enclave to decryptor enclave

The target enclave, with enclave and signer measurement values $MREnc_T$ and $MRSig_T$ respectively, initiates a local attestation request to the decryptor enclave. During the local attestation, the decryptor enclave ensures that the initiating enclave's signer measurement is the same as the one it received from the verifier enclave, that is, $hash(PK_V)$, and terminates in case it is not. If this check succeeds, the decryptor enclave concludes this local attestation handshake to establish the symmetric encryption key k_{DT} with the target enclave. It also generates a short-term keypair (pk_D, sk_D) and a token M.

The token M consists of the enclave measurement of the verifier enclave concatenated with a hash of the privacy policy that the verifier enclave sent and the decryptor's short-term public key: $M = MREnc_V||Hash(PP)||pk_D$. The decryptor enclave signs the token M using its long-term signing key $\mathcal{SK}_D$ and concatenates the token and the signature to form the token $H = M||Sig_{\mathcal{SK}_D}(M)$. It then encrypts the token H under k_{DT} and sends it to the target enclave. The decryptor now waits for further data along the secure channel with the signer measurement.

4.2.3 Runtime

In the runtime stage, the decryptor and target enclaves need to be online and one or more clients may attempt to connect to the website deployed within the target enclave. First, the target enclave sends the token H with each HTTP GET or POST request to a page with forms on the website. With a page load, the client checks for and retrieves the token H and verifies its integrity. For each Mitigator-supporting website, Mitigator's client is responsible for encrypting form fields to a decryptor enclave. It sends these encrypted fields to the target enclave. The target enclave then sends these fields to the decryptor enclave over the secure channel established through local attestation. The target enclave expects the decryptor enclave to return plaintext form fields over the same channel. We discuss communication between the client and the target enclave in the first subsection below and subsequently discuss communication between the target and the decryptor encalve. We also illustrate the entire exchange of messages that occur in this stage in Figure 4.4.

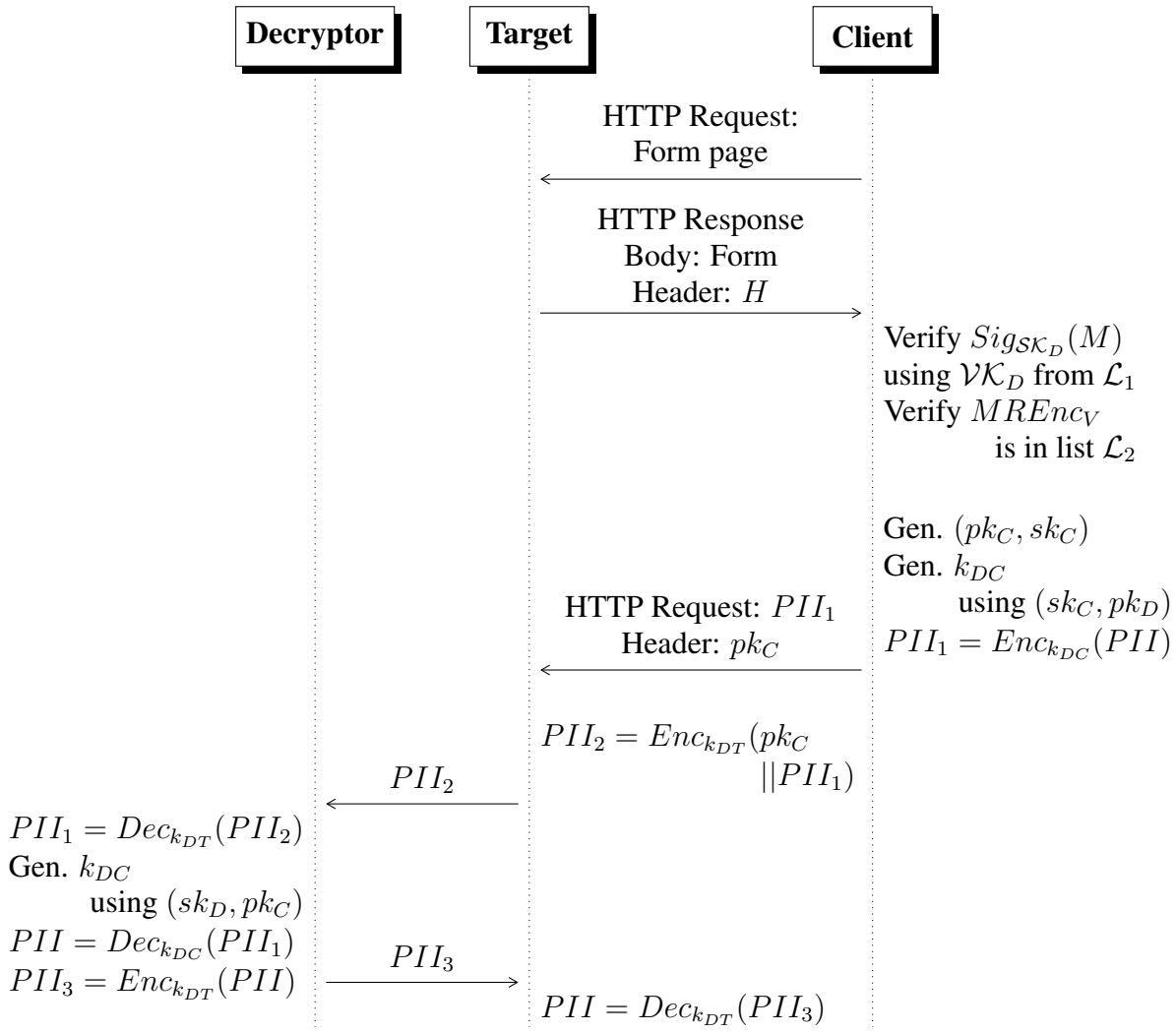

Figure 4.4: This figure captures the runtime stage of Mitigator. Prior to this stage, the target enclave should have conducted local attestation with the decryptor enclave resulting in the symmetric key k_{DT} being established between the two enclaves. The target enclave should have obtained the signed token H from the decryptor enclave over the secure channel established with it. This token has the form shown in Figure 4.3. We remark that the target enclave should send the token H in the response header of all HTTP responses to requests for an HTML form page. In order to verify the integrity of the token H, the client obtains the decryptor enclave's long-term verification key $\mathcal{VK}_D$ from list $\mathcal{L}_1$ and the verifier enclave's expected enclave measurement from list $\mathcal{L}_2$.

Client to target enclave

The client performs three checks on the integrity of the signed token H as follows. First of all, it checks that this token consists of another token M concatenated with a signature over M, such that the signature can be verified using the decryptor's long-term verification key $\mathcal{VK}_D$. If this check fails, then the adversarial OS on the remote host would have modified the token in transit or in general, sent an entirely different token and a signature over it with its own signing key. If the check succeeds, then the client expects the internal token M to contain the following values concatenated together: the verifier's enclave measurement, a hash of the privacy policy, and the decryptor's short-term public key.

In the second check, the client verifies that the enclave measurement contained in that token corresponds to that of a genuine open-sourced verifier enclave. If the second check fails, then the adversarial OS has attempted to run a malicious or lazy verifier, which, for instance, simply signs over the target enclave on the server side. In other words, the website source code cannot be said to have been checked for compliance if the second check fails. Finally, the client should check that the hash of the privacy policy displayed on the website is the same as that included in the token M. Again, if this check fails, then the adversarial OS may have returned a different privacy policy than the one that the verifier enclave checked the source code against. Specifically, the adversarial OS may attempt to return a more restrictive privacy policy, which may in turn give the users a false sense of security.

To facilitate the first two checks, the client contains two lists: a list $\mathcal{L}_1$ of hard-coded long-term verification keys of decryptors for each Mitigator-supporting website URL, and a list $\mathcal{L}_2$ of valid verifier enclave measurements. We describe in Section 4.2.4 how the client can obtain, in a trustworthy manner, the expected enclave measurement of the verifier and decryptor enclaves and the verification key of the decryptor enclave behind each Mitigator-supporting URL. The last check, however, does not require any additional state to be maintained by the client. The client may simply compute the hash of the displayed privacy policy and compare it to that in the signed token. The client proceeds to securely obtain the users' plaintext data and encrypt it to the decryptor enclave only if all three of these checks succeed. The specific technique used to securely obtain the users' plaintext data may vary across implementations; we detail our specific strategy in Section 5.6. If any check fails, then the client treats the website as if it did not support Mitigator.

The client first generates its own short-term keypair, say (pk_C, sk_C). It then uses the decryptor's short-term public key pk_D and its own short-term private key sk_C to generate the shared secret and derive a symmetric key k_{DC}. This key is subsequently used to encrypt form field values, which may contain users' personally identifiable information and are thus denoted by PII, whenever a user presses a button to submit the form. The client sends the resulting ciphertext

values, say PII_1, instead of the plaintext values in the subsequent HTTP request to the website. The client also sends its newly generated short-term public key pk_C in the above HTTP request. This enables the decryptor to derive the shared secret and symmetric encryption key k_{DC}.

Target enclave to decryptor enclave

The target enclave obtains the client's short-term public key pk_C with each HTTP GET or POST request by the client for pages with form fields. It also obtains HTML form field data after performing decryption with the TLS session key, as usual. However, for Mitigator-supporting websites, these HTML form fields are not plaintext data. They are ciphertexts that are encrypted to the decryptor's short-term public key for each client, pk_D. The target enclave then prepends each of the form fields with pk_C and encrypts the resulting data block to the decryptor's symmetric key that was established with the decryptor enclave at the time of local attestation, namely k_{DT}. It sends this ciphertext, say PII_2, to the decryptor enclave and waits for a response from the decryptor enclave.

Upon obtaining any data from the target enclave, the decryptor enclave removes the first layer of encryption, using the symmetric key k_{DT}. It then obtains the ciphertext form fields sent by the client (PII_1), along with its short-term public key, pk_C. The decryptor enclave first generates the shared secret and derives the symmetric encryption key k_{DC}, using its own private key sk_D and the client's public key pk_C. Using this key, it decrypts the ciphertext field PII_1 to obtain the plaintext form field PII. Finally, the decryptor enclave re-encrypts each such form field to the target enclave, using the key k_{DT}. It sends the resultant ciphertext, say PII_3 to the target enclave. Upon receiving the ciphertext PII_3 from the decryptor enclave, the target enclave simply decrypts it using the key k_{DT} to obtain the plaintext form field data. We remark that all encrypted messages between enclaves (PII_2, PII_3) as well as between the decryptor enclave and the client (PII_1) are IND-CCA2 (AEAD) authenticated encryptions.

4.2.4 Verifying the integrity of enclaves from the client

The protocols shown in Figures 4.3 and 4.4 do not require the verifier's or the decryptor's signer measurement value to be compared with another value on the server or client-side components. This allows the enclaves to be signed by the website provider and consequently, would make the process convenient for deployment on the server side. In other words, they do not need to be signed by a trusted third party. Given that these enclaves can be signed by the website provider, we proceed to explain below how the client can, nevertheless verify their integrity.

Verifying the integrity of the decryptor enclave

The client needs to establish the decryptor's long-term verification key to populate its list $\mathcal{L}_1$. To do this, it must first establish a secure channel with the decryptor enclave, authenticate it and then obtain the verification key over this secure channel. In order to establish a secure channel with each such decryptor enclave, the client needs to perform remote attestation with it and ensure that its enclave measurement in the remote attestation quote matches that obtained by compiling the open-sourced version. After doing so, the client can obtain its long-term verification key over the secure channel established through the attestation. The client needs to perform remote attestation with the decryptor enclave behind each Mitigator-supporting website each time the verification key changes. We repeat that the verification key should change infrequently; in fact, it should only change when the decryptor enclave is updated or if it is moved onto another machine.

The task of regularly performing remote attestation and maintaining the most recent verification key from every Mitigator-supporting website can consume a non-trivial amount of computational resources and network bandwidth for a Mitigator client. The client may therefore delegate this task to a privacy advocate organization. The organization can simply publish a list of websites and the verification key $\mathcal{VK}_D$ behind each Mitigator website. The client now simply fetches this list each time it fails to verify the signature in the header token H and attempts to re-verify it with an updated verification key, if any.

Verifying the integrity of the verifier enclave

Assuming that the trusted Intel SGX hardware would compute the enclave measurement of any enclave correctly, the enclave measurement of a lazy or malicious verifier enclave would be distinct from that of a correct verifier. Therefore, the client would detect a malicious verifier from its enclave measurement value, in the signed token H. Mitigator expects any valid verifier enclave to be made open-source as this enables experts to go through the source code and ensure that it actually performs relevant checks on the website's source code. This also implies that the client in Mitigator does not need to place implicit or unfounded trust in the verifier. The expected enclave measurement for the open-sourced verifier enclave can then be computed by compiling the source code for the remote platform on which the verifier enclave for the given website runs. A privacy advocate organization can be delegated to compile such open-source verifier enclaves for various platforms and maintain a list of valid verifier enclave measurements. The client populates its list $\mathcal{L}_2$ from such a list. In doing so, the client trusts the privacy advocate organization to publish the source code of various verifier enclaves online and ensure that experts appropriately check the soundness of the verifier enclave in checking the compliance of the source code with the privacy policy.

4.3 Security analysis

As mentioned in Section 4.1, our threat model includes an adversarial OS who wishes to perform operations on users' data that are not permitted as per the privacy policy. In this subsection, we explore such attacks. We remark that this is not a formal proof of security, which is beyond the scope of this work. We show that they may either be detected by Mitigator or that the OS is only limited to performing denial-of-service (DoS) attacks on its own Mitigator deployment. As the organization hosting these machines has incentives to deploy Mitigator, we presume that it would identify and eliminate sources of these DoS attacks. First, we argue that deploying a lazy or malicious decryptor enclave to perform local attestation with genuine verifier and target enclaves amounts to DoS attacks. For instance, a simple DoS attack could proceed as follows: the adversarial OS makes the verifier enclave perform local attestation with and send the signer measurement to a lazy or malicious decryptor enclave. However, following remote attestation with the decryptor enclave, the client would detect that its enclave measurement is not what the client expects it to be and this thus results in a DoS attack. Similarly, if the target enclave were to perform local attestation with a lazy or malicious decryptor enclave, then as the client would detect such an enclave from its enclave measurement following remote attestation, it would not encrypt any plaintext data to it.

We argue how Mitigator provides its guarantee that the decryptor enclave only allows target enclaves that have been signed by the verifier enclave to obtain clients' plaintext data. In the case when the genuine verifier enclave has not been run successfully to completion even once, we discuss the following attacks. Suppose that the OS attempts to sign the non-compliant target enclave with its own keypair, say $(\mathcal{SK}'_V, \mathcal{VK}'_V)$. As the decryptor does not authenticate the first enclave that successfully performs local attestation, an adversarial OS may then attempt to run a lazy verifier enclave. This lazy verifier enclave simply performs local attestation with the decryptor enclave and passes it the signer measurement of the non-compliant target enclave. However, the enclave measurement of this malicious enclave will be different than that of a genuine verifier enclave as the latter contains different code. As we discussed in section 4.2, the client checks that the enclave measurement reported in the signed token M, which is sent by the target enclave on behalf of the decryptor enclave, is that of a genuine verifier enclave. Therefore, when Mitigator is under the above attack, the client would detect the attack as it would obtain an enclave measurement $MREnc'_V$ that is not equal to that of a genuine verifier enclave. To evade detection, the adversary is left with only one choice: to let the decryptor enclave conduct local attestation with a genuine verifier enclave.

The OS may attempt to run the genuine verifier enclave once until it finishes local attestation, and then attempt to modify the ciphertext signer measurement sent by it, to an encryption of the signer measurement of the non-compliant target enclave. Within our threat model, the adversarial

OS cannot break assumptions for public key cryptography to obtain the corresponding shared secret between the decryptor and the verifier enclaves, or, to modify the message without the decryptor enclave detecting the attack. Finally, in the case when the verifier enclave has been run once until it writes a sealed keypair to disk, the adversarial OS may attempt to unseal this keypair and sign the non-compliant target enclave with it. We note that if the OS can successfully do so, then it can obtain the client's plaintext data and misuse it, without the decryptor enclave or the client detecting this attack. These systems would be under the impression that the compliant target enclave is being run as the enclave measurement of the target enclave is not used in our protocols. We explain in the sealing subsection below so as to how any attempts to unseal the keypair will only result in, at most, a denial of service attack against Mitigator.

We have discussed how an adversarial OS cannot spawn a non-compliant target enclave to obtain client's plaintext data. It could, however, attempt to obtain client's ciphertext data before it reaches the target enclave, modify or replay it. This is possible only if the TLS connection was terminated outside the enclave. We discuss below how the adversarial OS cannot gain anything from doing so. Moreover, the adversarial OS cannot modify encrypted messages from the target enclave to the decryptor enclave for the same reason that it cannot modify messages between the verifier and the decryptor enclave, which we discussed above. Finally, it cannot gain anything from replaying these messages, as we discuss below. We conclude this section with an analysis of how the security guarantees of our design hold as the enclaves are being updated.

4.3.1 TLS termination

The current target enclave design allows a TLS master certificate to be generated by the untrusted operating system. The TLS master private key can be kept outside the enclave and therefore, the TLS termination itself could occur outside the enclave. In case there is no secure channel between the Apache server enclave and untrusted code that does TLS termination, or, when an adversarial OS can *also* decrypt the TLS connection, then the adversarial OS can obtain, manipulate and replay ciphertext form fields and the client's short-term public key. We first discuss what happens when the OS tries to modify these fields.

In case the adversarial OS generates a keypair (sk_M, pk_M) and replaces the client's key pk_C with the public key pk_M, then the decryptor enclave will return incorrect plaintext upon decryption of the ciphertext fields. Similarly, as the adversarial OS cannot break the assumptions for the symmetric key cryptography algorithm used to encrypt data after local attestation, it cannot modify the ciphertext sent by the client, PII_1 to another one without also modifying the corresponding tag and thereby alerting the decryptor enclave. Therefore, for a successful attack, it must present data that has also been encrypted under a shared secret derived from the private

key of one of the enclaves and the public key of the other. As we assume that the browser has not been modified to exfiltrate the client's short-term private key sk_C, the adversarial OS cannot obtain it. Secondly, as the EPC of the remote machine on which the decryptor enclave is being run, is isolated from the OS, the OS cannot obtain sk_D. Under our adversarial model, without either of these, the OS cannot obtain the shared secret as we assume that it cannot break public key cryptography primitives. Therefore, the OS cannot obtain the client's plaintext PII, modify it to PII', and re-encrypt it to the target enclave. Of course, modifying random bits in the ciphertext can easily be detected by the target enclave as the tags would not match. In other words, the adversarial OS is only left with a possible replay attack of a public key and corresponding ciphertext data that were sent by a client previously. An adversarial OS whose intent is to replay the same ciphertext to initiate a false transaction on behalf of the user is out of our threat model. However, the decryptor enclave may protect against such an attack by recording all ciphertexts (PII_1) that it received and rejecting any repeated ciphertexts.

4.3.2 Sealing

Sealing is used in Mitigator to store the long-term signing-verification keypairs, generated by the verifier and the decryptor enclaves, to disk. If an adversary can access the plaintext of a keypair file sealed by the verifier enclave, then it can generate a signature over other malicious enclaves. Our threat model does not permit an adversarial operating system to simply obtain an unsealed file. It can, however, attempt to run an enclave that then attempts to unseal the keypair file and exfiltrate the keypair. We discuss this attack, a file deletion attack, and a file replay attack below.

Suppose that the adversarial OS attempts to run an enclave A in order to exfiltrate the verifier's keypair. We remark that the keypair files are sealed using the enclave measurement of the respective enclaves as input to the sealing key, rather than the signer measurement. Therefore, only enclaves that contain the same code as the given Mitigator enclave can successfully unseal the keypair. Therefore, for enclave A to successfully unseal the keypair and obtain it in plaintext form, it must contain the same code as the verifier enclave. Given that the verifier enclave only outputs the keypair in a ciphertext form by sealing it under its enclave measurement, enclave A cannot be executed in any way to exfiltrate the keypair.

This also implies that, for instance, enclave A cannot execute code that reseals the keypair under its signer measurement, because if this code was included in its executable then it would have a different enclave measurement. Therefore, in turn enclave A would fail to unseal the keypair in the first place. Additionally, as enclave A is bound to contain the same code as the verifier enclave, it would only sign the target enclave and conduct its local attestation handshake with the decryptor enclave if the compliance check succeeds. In this case, the local attestation

handshake would be identical to that of the verifier enclave, except possibly with the signer measurement of enclave A being different than that of the verifier enclave. However, the signer measurement of the verifier enclave is not used within the protocol shown in Figure 4.3 and thus cannot affect its correctness. Moreover, enclave A can only at most attempt to send the correct signer measurement of the target enclave to the decryptor enclave. An identical argument can be made for this attack on the decryptor enclave: the malicious OS-spawned decryptor enclave can only contain code that is identical to a genuine decryptor enclave. It will therefore, only send the enclave measurement of a valid verifier enclave, if the verifier enclave attested to it.

We have therefore established Mitigator's security in face of the above attack where a masquerading enclave attempts to unseal a Mitigator enclave's sealed keypairs. We proceed to discuss file deletion and replay attacks against the sealed keypair files. The verifier enclave is designed to generate and seal a new keypair each time it is updated. This keypair is then used to sign the target enclave. If the verifier enclave's keypair file is repeatedly deleted by the adversarial OS, then in each execution, the verifier enclave simply generates a new signing keypair to sign the target enclave. On the other hand, when a malicious operating system replaces a newer keypair with an old keypair, we have a file replay attack. When the verifier enclave is under a file replay attack, it now signs the target enclave with an old key and communicates the signer measurement of the target enclave, based on this old key, to the decryptor enclave.

Similar to the verifier enclave, the decryptor enclave also generates and seals a new keypair each time it is updated. This keypair is used to sign the token H to be sent to the client. If the sealed keypair file of the decryptor enclave is repeatedly deleted and replaced with a new one, then the enclave simply keeps generating and sealing new keypairs. The most successful attack simply results in a DoS: if the most recent verification key is not reflected in the browser extension's list $\mathcal{L}_1$, then the extension would not be able to verify the authenticity of the token H, thereby leading to the user rejecting Mitigator's support for the website. Whereas, if the extension does obtain the most recent key, then the rest of the protocol works as in the case when the system is not under attack. When the decryptor enclave is under a file replay attack, it signs the token H with an old key. If the browser extension has the new key, then it would again fail to verify the token H, resulting in a DoS attack on Mitigator. If it has the older key, then the rest of the protocol functions as usual.

Therefore, both file deletion and replay attacks against the decryptor enclave may, at best, result in the rejection of an authentic target enclave by the browser extension and therefore, poor user experience. Importantly, none of these attacks against the verifier or target enclave result in a non-compliant target enclave obtaining user data; that is, Mitigator is sound in the face of these attacks. As we assume that the old signing key may not leak outside the enclave, we do not consider file replay attacks in our threat model. Ultimately, these attacks are a form of a

denial-of-service attack and we do not defend against this general class of attacks in our threat model; a certificate revocation scheme can be used to address this.

4.3.3 Security while updating enclaves

Updating the target enclave

Following any update to the website server source code, an adversary would require a signature over the updated target enclave, with the verifier's long-term signing key $\mathcal{SK}_V$. This is because an authentic decryptor enclave would check that the signer measurement of a target enclave matches the hash of $\mathcal{VK}_V$, before handing over plaintext client data. Now, any adversary, other than the verifier enclave with the correct enclave measurement value, cannot unseal the keypair. Therefore, it can not produce a signature over the target enclave that passes the check by the decryptor enclave. Therefore, following each update to the website server source code, the verifier must verify and resign the target enclave, before it can obtain users' plaintext form field data.

Updating the verifier enclave

We foresee updates to the verifier enclave being pushed from a central online repository and the website provider now needs to update their target enclave source code such that it passes a check by the updated verifier enclave, within a reasonable time interval T of an update in the verifier's source code. (Alternatively, the verifier enclave measurements may be published on this repository with an expiry date, such that any verifier enclave measurements which are expired are considered as invalid, thereby resembling a certificate PKI system.) Thus, the browser extension may be expected to only accept a token H that contains the enclave measurement of an updated enclave and not of an older one. The decryptor enclave generates this signed token using the enclave measurement of the enclave that conducts the first successful local attestation. It is therefore necessary that after each update in the verifier's source code, it should recheck the compliance of the target enclave's source code, even if it has not been updated. Following a successful check, it should perform local attestation with the decryptor enclave as usual.

Second, with every update in the verifier's source code, the contents of the message that it sends to the decryptor enclave following local attestation will change, as we explore below. As mentioned previously, the long-term verification keypair $(\mathcal{SK}_V, \mathcal{VK}_V)$ of the verifier enclave is sealed to disk using its enclave measurement. Consequently, in normal operation such a sealed keypair would not be accessible after any update to the verifier source code. We remark that when sealing under the enclave measurement, following any modifications to the enclave code,

including any security updates that result in a changed Security Version Number (SVN), unsealing is not successful. Whenever the long-term keypair cannot be unsealed successfully, a new keypair is generated. Therefore, with every update in the verifier source code, a new long-term keypair will be generated and used to sign the target enclave. Consequently, whenever the verifier enclave conducts local attestation with the decryptor enclave following an update in the verifier's source code, it will send a hash of the new verification key. In turn, this enables the decryptor enclave to correctly verify a resigned target enclave.

Therefore, following any updates to the verifier enclave, as long as the website provider reruns the verifier and the target enclaves, such that by time T after the verifier enclave update, both of the following tasks occur, then any clients who send requests to the target enclave after that time will obtain the updated token H. Both verifier and target enclaves are to perform local attestation with the decryptor enclave[3] and the latter must send the updated token to the target enclave, within time T after the verifier enclave update. In other words, an update in the verifier will be reflected in an updated token after Mitigator goes through another run of the setup shown in Figure 4.3.

Updating the decryptor enclave

As was the case with the verifier enclave, we observe that as the decryptor's long-term keypair is sealed to disk using its enclave measurement value as input to the sealing key, it cannot be unsealed after an update in the decryptor's source code. This keypair is therefore resampled with every such update. Therefore, following a remote attestation with an updated decryptor enclave, the client would always obtain a new verification key. Finally, since the decryptor enclave simply waits for local attestation following a successful generation of its long-term keypair, effectively an update in the decryptor will also require the verifier and target enclaves to perform local attestation with it again.

Second, the Mitigator client or a trusted privacy advocacy organization should perform remote attestation with the decryptor enclave, authenticate it and obtain the updated verification key $\mathcal{VK}_D$ over the resulting secure channel. We remark that in case the browser extension does not obtain the updated verification key, then it may reject a genuine, updated decryptor enclave, thereby degrading the user's experience of a Mitigator-supporting website. It does not, however, lead the extension to falsely accept any malicious decryptor enclave as a genuine one.

[3] In our implementation, the decryptor enclave must be restarted in order to allow the verifier and target enclaves to perform local attestation with the decryptor enclave. A more sophisticated implementation can support the local attestation facilities without requiring a restart and without compromising the guarantee that we show in Section 5.1

4.4 Design choices

In this section, we discuss some design choices that are relevant to the deployability of Mitigator and to achieving security within its threat model.

4.4.1 Distributing enclaves across server-side machines

We note that all guarantees provided by Mitigator will be preserved in case the decryptor enclave is on a different machine as compared to both the verifier and the target enclave. The only difference to the architecture would be that the decryptor enclave would perform local attestation with these enclaves in case it is on the same machine as them and remote attestation in case it is not. The verifier and the target enclaves may be on different machines, as long as the verifier can access the website source code files and it contains or is passed enough input arguments to compile an enclave image for the machine on which the target enclave is to be executed. Additionally, we observe that the current design of Mitigator assumes a single target enclave process, in that the decryptor enclave stops listening for local attestation requests after obtaining the second such request, which would correspond to the target enclave. However, if it were to continue listening for other local attestation requests, the decryptor enclave can establish a secure channel with multiple target enclaves, through the local attestation. Each decryption request from an authenticated target enclave process would then be treated independently of requests from other such processes.

4.4.2 Choice of sealing key input

In Section 4.3.2 we discussed the security of Mitigator against an attack wherein an adversarial OS runs another enclave to extract keypair files that have been sealed by a genuine Mitigator enclave. The fact that the keypairs were sealed to disk using the enclave measurement and not the signer measurement limits the code that our adversarial OS can run from within the enclave to exfiltrate the keypair to that of a genuine Mitigator enclave. If the signer measurement of the verifier enclave was used, instead of its enclave measurement, as input to the sealing key, then the following masquerading attack goes undetected by the browser extension. We observe that two relevant choices in Mitigator's current design enable this attack: first, the decryptor sends the verifier's *enclave measurement* value to the browser extension and second, the verifier and decryptor enclaves could be signed by the website provider.

An adversarial OS simply generates and signs a malicious or lazy verifier, $\mathcal{V}_m$, using ($\mathcal{SK}_A$, $\mathcal{PK}_A$) and it has two target enclaves: a compliant one and a non-compliant one. The malicious

and the genuine verifiers differ in their enclave measurement values, as they have been compiled from different source code, but have the same signer measurement value as they have been signed using the same key. The genuine verifier $\mathcal{V}_g$ is run with the compliant target enclave source code as input. It creates and seals a long-term signing, verification keypair to disk, say ($\mathcal{SK}_V$,$\mathcal{VK}_V$), using its signer measurement value, namely the hash of $\mathcal{PK}_A$, as input to the sealing key. It then signs over the compliant target enclave and then conducts local attestation with the decryptor enclave to notify it of the hash of the verification key $\mathcal{VK}_V$. Upon execution of the malicious verifier, say $\mathcal{V}_m$, it successfully unseals this keypair as it was sealed using its signer measurement, which would again be the hash of $\mathcal{PK}_A$. It is passed a copy of the non-compliant website source code as input and it signs over it, using $\mathcal{SK}_V$. Now, the non-compliant target enclave is executed: it will pass the authentication check by the decryptor enclave as it would also have been signed by $\mathcal{SK}_V$. The decryptor will send in the enclave measurement of the authentic verifier enclave $\mathcal{V}_g$ to the browser extension and therefore, the extension will not detect this attack. The adversarial OS succeeds if it gets access to the long-term signing key $\mathcal{SK}_V$ of the verifier enclave, which it can easily do in case this key is sealed with the verifier's signer measurement instead of its enclave measurement. Furthermore, there is no way to prevent this attack unless the website provider is trusted to not reuse the key that is used to sign the verifier enclaves to also sign other possibly malicious enclaves, which may simply extract the key $\mathcal{SK}_V$.

This attack is stealthy as the decryptor enclave sends to the client a measurement that is the same across both malicious, genuine verifier enclaves (signer measurement) and not one which differs across the two (enclave measurement). Therefore, it is evident that in a scheme wherein the sealing keys for keypair files for the verifier and decryptor enclaves are derived from the signer measurement, in order to *detect* the above attack, the enclave measurement of the verifier enclave be sent to the client in the signed token H. Consequently, the client is then required to trust the key used by the website provider to sign the Mitigator enclaves. The attack cannot be *prevented* unless this website provider is restricted from using this key to sign other enclaves. This is because allowing the verifier and decryptor enclaves to be signed by the website provider, that is, an adversarial OS *and* sealing their keypair files to disk under the signer measurement, effectively allows the OS to use the keypair used to sign these enclaves to then sign malicious enclaves that can *also* unseal such keypair files. It may be unreasonable to simply trust the website provider to follow this restriction.

4.4.3 Comparison to related work

As discussed in Section 3.4, Birrell et al.'s [BGR+18] use-privacy-based trusted hardware platform scheme is similar to ours in its aim. With an understanding of our proposed design, we proceed to contrast their designs with ours. Their delegated monitoring scheme differs from their

source-based monitoring scheme, which we described in Section 3.4, in that instead of having the reference monitor placed on the same machine as the data source and mediating multiplexed requests from multiple applications, it is placed on the same machine as the target application and it relays requests on behalf of target applications to the data source. Their inline monitoring scheme involves modifications to the target application to request for data items from the data source. Thus, our design is closer to their delegated monitoring scheme than their source monitoring scheme: our decryptor enclave plays the roles of both the data source and the reference monitor, but in our current design, only responds to requests for decryption of client ciphertext data from a single target enclave. In comparison to their inline monitoring scheme, which involves changing the target application source code when it requires data to interact with an extended API for their reference monitor, our target enclave only needs to be modified slightly to request for client data from the decryptor enclave. We provide a very small interface (currently one function) for performing this request. Thus, their delegated monitoring scheme is closest to our design. However, there are some significant differences.

Importantly, in the absence of further specifications, it is unclear whether the data source authenticates the monitor based on its enclave measurement or its signer measurement and therefore, as we have seen in Section 3.4, the corresponding trusted computing base required for the monitor to work correctly in the face of an untrusted OS, is unclear. In Mitigator, the users do not need to trust that the website provider owns any additional keypairs as the remote enclaves, in our design, could be signed with any keys. Furthermore, in Section 4.4 we include an analysis of the repercussions of using enclave and signer measurements as input to the sealing key for remote enclaves in our design.

Second, the monitor authenticates the *untrusted application* and not the target enclave, to use the data and then hands over the plaintext data to this application. The monitor mediates messages between the untrusted application and the target enclave, to authenticate the *enclave* before it obtains any user data; this check still allows the untrusted OS to observe the plaintext user data that the application obtained from the monitor and thereby defeats the purpose of using the trusted hardware platform. Similarly, the specification of the design of the source monitoring scheme does not include whether data items are encrypted back to the target enclave or not. However, the system allows for this possibility.[4] In contrast, in our design, an untrusted application never obtains plaintext data: we ensure that only the target enclave obtains users' data from the decryptor enclave, as the latter encrypts it to a key derived from a local attestation shared secret with the former.

[4] Assuming that the target enclave includes a public key in its quote, the monitor may send back a public key and encrypt the data item to the enclave, using symmetric-key encryption.

4.5 Summary

In this chapter, we have presented our main design. Our design consists of a *verifier* enclave that checks the source code of the target enclave for compliance with the privacy policy and if the target is found to be compliant, the verifier enclave signs it and informs another *decryptor* enclave of the expected signer measurement of the target enclave through attestation. The *decryptor* enclave verifies the identity of the target enclave through its attested signer measurement and ensures that only legitimate target enclaves can obtain the clients' form field data. We have also outlined the threat model that we seek to defend against, namely that of an adversarial OS. We have presented a security analysis of our system against this adversary and in particular, we describe how we provide the client a guarantee that their data will only be handled by code that is compliant with the displayed privacy policy, as checked by a valid verifier enclave. Finally, we have also emphasized the implications of certain design choices in achieving the above guarantee in the presence of our adversary and have discussed how our design compares to the closest related work.

Chapter 5

Implementation

In this chapter, we proceed to discuss how we implement the design proposed in Chapter 4. We begin with an in-depth analysis of the decryptor enclave implementation for both the post-verification and the runtime stages, and present the guarantees that it provides, in Section 5.1. We then outline how we modify the Graphene-SGX tool to support our design, and in particular, to run the verifier and target enclaves, in Section 5.2. We present the main algorithm followed by the verifier enclave, in Section 5.3. Here we describe how it uses an adapted source code analysis tool to check the target enclave source code and in turn, how the target enclave needs to be set up within the Graphene-SGX platform in order to support the above check. We also discuss how the verifier enclave generates a signature over the target enclave in that section. We proceed to describe how we adapted the Pixy source code analysis tool to conduct the above compliance check in Section 5.4. At runtime, our design involves interactions between the decryptor enclave and the target enclave, as well as between the target enclave and the client-side extension. Therefore, we discuss how the target enclave implements its interactions with the Mitigator's browser extension and the decryptor enclave in Section 5.5. We conclude with a brief description of how Mitigator's browser extension performs the requisite functionalities on behalf of the client at runtime in Section 5.6.

5.1 Decryptor enclave

The decryptor is executed as a native Intel SGX enclave. It consists of an untrusted decryptor application and a decryptor enclave. The decryptor enclave maintains state corresponding to the global variables shown in Algorithm 5.1. Apart from local attestation and sealing-related calls,

Table 5.1: Internal state maintained in the decryptor enclave. The last six variables are initialized to arrays of zeros of length given by the key sizes of the symmetric key cipher (lines 5–6), the asymmetric key cipher (lines 7–8), and the signature scheme (lines 9–10).

1: **global variables** ▷ Enclave global variables for ecalls for the decryptor enclave
2: $successful\_la\_count \leftarrow 0$
3: $verifier\_mr\_enclave \leftarrow [0]^{32}$
4: $target\_mr\_signer \leftarrow [0]^{32}$
5: $verifier\_session\_key$
6: $target\_session\_key$
7: $short\_term\_private\_key$
8: $short\_term\_public\_key$
9: $verification\_key$
10: $signing\_key$
11: **end global variables**

the decryptor enclave also consists of three other ecalls. The $ecall\_process\_verifiers\_message$ ecall processes a message from the first enclave that is successfully authenticated and is described in Algorithm 4. This ecall changes the internal state of the enclave, but does not return any values to the application. Next, the $ecall\_create\_and\_encrypt\_mitigator\_token\_H$ ecall returns the token H from Figure 4.3 encrypted to the target enclave, and is shown in Algorithm 5. Finally, the $ecall\_process\_targets\_message$ ecall processes a message from the second enclave that is successfully authenticated and generates a response to that message; it is depicted in Algorithm 6. We note that the decryptor enclave maintains internal state corresponding to the variables shown in Table 5.1, but these variables are not accessible by the untrusted main application.

The untrusted main application follows the algorithm presented in Algorithm 2. This algorithm calls two wrapper functions; the $local\_attestation\_as\_responder$ function calls ecalls to obtain or process local attestation messages. As the decryptor enclave acts as a responder enclave in its local attestation with both the verifier and the target enclaves, it contains ecalls to generate the first, third messages, and to process the second one. The ecall to process the second message, which contains the peer enclave's report, taps into the internal enclave function call $verify\_and\_set\_key$, shown in Algorithm 3, to authenticate the peer enclave based on its report. Local attestation messages themselves are passed over the given IPC channel. The second wrapper function, namely $decryptor\_set\_signing\_keypair$ follows Algorithm 1 and consists of ecalls to unseal an existing keypair and/or to generate and seal a new keypair.

We begin with a discussion of the algorithm followed by the untrusted main application. The untrusted application first calls an untrusted wrapper, shown in Algorithm 1, that calls

Algorithm 2 Main algorithm for local attestation and secure communication with the verifier and decryptor enclaves

```
1:  ▽ This is an untrusted application call.
2:  function DECRYPTOR_UNTRUSTED_MAIN(decryptor_keypair_path)
3:      ▽ Untrusted wrapper that calls into ecalls for setting up the enclave's long-term keypair.
4:      set_signing_keypair(decryptor_keypair_path)
5:      ▽ Untrusted wrapper that calls into ecalls for conducting local attestation as a responder
        enclave.
6:      successful_verifier_la ← local_attestation_responder(verifier_ipc_channel)
7:      if successful_verifier_la ≠ True then
8:          return 0
9:      end if
10:     ciphertext ← receive_message(verifier_ipc_channel)  ▷ Untrusted call.
11:     ecall_process_verifiers_message(ciphertext)
12:     successful_la_with_target ← local_attestation_responder(target_ipc_channel)
13:     if successful_la_with_target ≠ True then
14:         return 0
15:     end if
16:     encrypted_token_H ← ecall_create_and_encrypt_mitigator_token_H()
17:     send_message(encrypted_token_H, target_ipc_channel)  ▷ Untrusted call.
18:     while True do
19:         input_ciphertext ← receive_message(target_ipc_channel)
20:         output_ciphertext ← ecall_process_targets_message(input_ciphertext)
21:         send_message(output_ciphertext, target_ipc_channel)
22:     end while
23: end function
```

into ecalls for setting up the decryptor enclave's state of its long-term signing and verification keypair. The untrusted application then waits for a local attestation request from the verifier enclave. Upon obtaining a request, it completes the local attestation handshake through the $local\_attestation\_responder$ untrusted wrapper function (line 3). If the local attestation handshake and authentication is successful (lines 4–6), the main application proceeds to wait for and receive a ciphertext message from the verifier enclave (line 7). It then processes this message through the $ecall\_process\_verifiers\_message$ ecall (line 8).

The application then waits for a local attestation request from the target enclave and responds to it, through the $local\_attestation\_responder$ untrusted wrapper function (line 9). Following a successful local attestation handshake with the target enclave (line 10–12), it then ex-

Algorithm 3 Algorithm for an internal enclave functions that authenticates the peer enclave that performs local attestation, based on its enclave and signer measurements.

```
1:  ▽ This is an internal enclave call.
2:  function VERIFY_AND_SET_KEY(mr_enclave, mr_signer, session_key)
3:      if successful_la_count = 0 then   ▷ Take this to be the verifier enclave
4:          verifier_mr_enclave ← mr_enclave
5:          verifier_session_key ← session_key
6:          successful_la_count ← 0
7:      else
8:          for i ← 1, 32 do
9:              if target_mr_signer[i] ≠ mr_signer[i] then
10:                 return 0
11:             end if
12:         end for
13:         target_session_key ← session_key
14:         successful_la_count ← 1
15:     end if
16:     return 1
17: end function
```

ecutes the ecall $ecall\_create\_and\_encrypt\_mitigator\_token\_H$ to obtain the ciphertext token $encrypted\_token\_H$ (line 13). The main application then sends this token to the target enclave along an IPC channel (line 14). It then keeps waiting for further messages from the target enclave (line 16), processes any received message through the ecall $ecall\_process\_targets\_message$ to get a response $output\_ciphertext$ (line 17). It sends the response message back to the target enclave (line 18). This exchange of messages goes on until the application is stopped and corresponds to the decryptor enclave decrypting client's ciphertext form fields and re-encrypting the plaintext to the target enclave.

We have discussed the expected order of execution of ecalls by the main application, but since it is untrusted, it may perform these ecalls in an arbitrary order or a different number of times. We need to demonstrate that by doing so, the untrusted main application cannot subvert the intended functionalities of the decryptor enclave. To this end, we proceed with a brief description of the internal enclave function and each of the ecalls, in order to explain how each of these functions cannot be subverted in either of the two aforementioned ways. For each function, we conclude with a simple enumeration of guarantees that it provides.

verify_and_set_key: This internal enclave call, which is shown in Algorithm 3, authenticates the peer enclaves in a local attestation handshake based on their enclave, signer measurements and stores corresponding session keys for later use. First, it is important to note that this function will only be called by trusted enclave ecalls and in particular, it will be called with genuine measurements of a peer enclave, as per a verified report, and the session key for that peer enclave. In other words, it will only be called with genuine arguments, as many times as valid (possibly identical) reports are being presented in a local attestation handshake.

This function uses the $successful\_la\_count$ enclave global variable to keep track of the number of times it has been called. It does not compare its input arguments to any hard-coded values the first time it is being called. It instead stores the input enclave measurement into the $verifier\_mr\_enclave$ variable for use by other ecalls. It authenticates the second and subsequent enclaves to call this function by checking that its signer measurement is the same as that in the $target\_mr\_signer$ variable. It also sets the $verifier\_session\_key$ variable to the session key of the first enclave and the $target\_session\_key$ variable to the session key of subsequent enclaves. The $successful\_la\_count$ variable and the $verifier\_mr\_enclave$ variables are not set elsewhere in the code. Thus, it is evident that the following properties hold true:

1. The $successful\_la\_count$ variable correctly refers to the number of successful authentications of attesting enclaves. That is, it is 1 only after one enclave has been authenticated successfully and it is 2 only after two enclaves have been authenticated successfully.

2. The second enclave that has been successfully authenticated has a signer measurement that is equal to the $target\_mr\_signer$ variable.

3. If the $successful\_la\_count$ variable is 1 or more, then the $verifier\_mr\_enclave$ variable refers to the enclave measurement of the first enclave to call this function during local attestation.

4. If the $successful\_la\_count$ variable is 1 or more, the $verifier\_session\_key$ variable refers to the session key of the first enclave that has been authenticated successfully.

5. If the $successful\_la\_count$ variable is 2 or more, the $target\_session\_key$ variable refers to the session key of the latest enclave that has been authenticated successfully.

ecall_process_verifiers_message: This ecall first ensures that it is being called when the value of the $successful\_la\_count$ variable is 1 (line 2); that is, after a successful authentication of the verifier enclave has taken place. (If this is not the case, then the ecall does nothing.) It decrypts its input argument with the session key established with the verifier enclave, namely the key in the $verifier\_session\_key$ variable (line 3). It expects the resulting plaintext to contain

Algorithm 4 Algorithm for an ecall to process a message from the verifier enclave.

```
1: function ECALL_PROCESS_VERIFIERS_MESSAGE(ciphertext)  ▷ This is an ecall.
2:     if successful_la_count == 1 then
3:         (decrypt_status, plaintext) ← decrypt(verifier_session_key, ciphertext)
4:         if decrypt_status ≠ True then
5:             return "Error in decrypting: OS might have modified the message"
6:         end if
7:         (target_mr_signer, privacy_policy_hash) ← plaintext
8:     end if
9: end function
```

two hashes: the target enclave's expected signer measurement and a hash of the privacy policy files on which the compliance check was performed. It sets the variables $target\_mr\_signer$ and $privacy\_policy\_hash$ to these values. In other words, as long as statements 1 and 4 are correct, then we obtain the following:

6. Only after one enclave has been successfully authenticated, the $target\_mr\_signer$ variable is set to a part of the plaintext content of the first message sent by that enclave.

7. Only after one enclave has been successfully authenticated, the $privacy\_policy\_hash$ variable is set to a part of the plaintext content of the first message sent by that enclave.

We note that this function can be called multiple times with the same or different ciphertext messages. It is evident that subsequent runs of the function with identical inputs result in identical results as the first run. Given that the verifier enclave only sends one valid message to the decryptor enclave in each run of the post-verification stage of our system, the adversarial OS does not have any other valid messages to replay for that run. Any messages that were not encrypted to the shared secret result in a decryption failure (line 2) and consequently, do not result in a change of the internal state of the enclave.

ecall_create_and_encrypt_mitigator_token: This ecall first ensures that it is being called after two successful authentications have taken place, that is, the $successful\_la\_count$ variable is at least 2 (line 2). The ecall generates a short-term private-public keypair. It then concatenates the newly generated public key $short\_term\_public\_key$ with the two global variables $verifier\_mr\_enclave$ and $privacy\_policy\_hash$ to form the variable $token\_M$. It signs the resulting token M with its long-term signing key to obtain the token H. Finally, it encrypts the token H using the target enclave's session key, namely $target\_session\_key$. Therefore, given

Algorithm 5 Algorithm for an ecall to encrypt the signed token $\mathcal{H}$ to the target enclave.

1: **function** ECALL_CREATE_AND_ENCRYPT_MITIGATOR_TOKEN_H() ▷ This is an ecall.
2: **if** $successful\_la\_count \geq 2$ **then**
3: $(short\_term\_private\_key, short\_term\_public\_key) \leftarrow create\_keypair()$
4: $token\_M \leftarrow short\_term\_public\_key \| verifier\_mr\_enclave \| privacy\_policy\_hash$
5: $token\_M\_signature \leftarrow sign(long\_term\_signing\_key, token\_M)$
6: $token\_H \leftarrow token\_M \| token\_M\_signature$
7: $encrypted\_token\_H \leftarrow encrypt(target\_session\_key, token\_H)$
8: **return** $mitigator\_encrypted\_token\_H$
9: **else**
10: **return** "INVALID ECALL"
11: **end if**
12: **end function**

the pseudocode for this function, we obtain the following statement on the basis of statements 3 and 7 above, or, alternately, on the basis of statements 1, 3, and 4:

8. Only after at least two enclaves have been successfully authenticated, the $token\_H$ variable, which is sent later to the client, contains the enclave measurement of the first enclave to attest to the decryptor enclave and the privacy policy hash, which was in the first message sent by that enclave, signed with the decryptor enclave's long-term signing key.

We observe that if this function is called multiple times, then it must be called after two attesting enclaves have successfully been authenticated (line 2) for the function to return with a new token. The enclave measurement of the first enclave to be successfully authenticated and the privacy policy hash do not change between the first and subsequent calls to this function, as per statements 2 and 7 above. Therefore, the tokens returned in the first and subsequent calls only differ in the short-term public key variable, which is desirable as this key is supposed to be ephemeral.

Finally, we observe that the untrusted wrapper $set\_signing\_keypair$ and the resulting ecalls can be called at any point before the above $ecall\_create\_and\_encrypt\_mitigator\_token\_H$ ecall is called. Not calling this wrapper at all or calling it after the above ecall has been called would result in a failure in the construction of a token H and thus would simply result in a DoS attack on our design.

$\boldsymbol{ecall\_process\_targets\_message}$: This ecall first ensures that it is being called after two successful authentications have taken place; that is, the $successful\_la\_count$ variable is at

Algorithm 6 Algorithm for an ecall to process a message from the target enclave and to respond to it with another message

1: **function** ECALL_PROCESS_TARGETS_MESSAGE($input\_ciphertext$) ▷ This is an ecall.
2: **if** $successful\_la\_count == 2$ **then**
3: $(decrypt\_status, message) \leftarrow decrypt(target\_session\_key, input\_ciphertext)$
4: **if** $decrypt\_status \neq True$ **then**
5: **return** ("Error in decrypting main message", $input\_ciphertext$)
6: **end if**
7: $(client\_public\_key, ciphertext\_form\_data) \leftarrow split\_message(message)$
8: $client\_session\_key \leftarrow derive\_key(short\_term\_private\_key, client\_public\_key)$
9: $(decrypt\_status, plaintext\_form\_data) \leftarrow decrypt(client\_session\_key, ciphertext\_form\_data)$
10: **if** $decrypt\_status \neq True$ **then**
11: ▽ Return back the same data that was encrypted to it.
12: $plaintext\_form\_data \leftarrow ciphertext\_form\_data$
13: **end if**
14: $output\_ciphertext \leftarrow encrypt(target\_session\_key, plaintext\_form\_data)$
15: **return** $output\_ciphertext$
16: **else**
17: **return** "Invalid ecall"
18: **end if**
19: **end function**

least two. This ecall implements the target enclave's processing of client data in Figure 4.4. That is, it decrypts its input argument using the target enclave's session key, establishes a shared key with the client, decrypts the form field data with this shared key and then re-encrypts it with the session key for the target enclave. Thus, we arrive at the following statement on the basis of statements 1, 2, 5, and 6 above:

9. The client's form field data is encrypted to an enclave whose signer measurement was sent by the first enclave to attest to the decryptor enclave.

We remark that in case the decryptor enclave cannot successfully decrypt the client's ciphertext form data (line 10), it re-encrypts this data to the target enclave and returns it. This could occur if the client erroneously sent plaintext form field data or used the wrong public key. Finally, we observe that if a valid target enclave passes valid client ciphertext form field data, it always obtains the users' plaintext data. In particular, if the target enclave's form webpage displays

user's plaintext data, possibly following modifications, back to them in a webpage, it is essential that the decryptor enclave should not have a reference monitor-like design in that it should not refuse to decrypt any data requested by a valid target enclave. Designs of the decryptor enclave that allow it to refuse decryption of ciphertext client data, say sent from specific webpages, from a valid target enclave face a significant usability problem. That is, the users will now unexpectedly fail to view any data on other webpages that is derived from the filled-in form fields. We therefore believe that such a reference monitor-like design will not be acceptable to end users.

Now, given that the client side extension is to check the enclave measurement included in the token M that it receives is that of a valid verifier enclave, statement 8 implies that the extension will ensure that the first enclave to attest to the decryptor enclave is a valid verifier enclave. Therefore, statement 9 is equivalent to the following: the client's form field data is encrypted to an enclave whose signer measurement was sent by a valid decryptor enclave and thus, the former enclave is a valid target enclave. Additionally, as the client side extension also checks that the hash of the displayed privacy policy is the same as that included in the token H, the valid verifier enclave would have checked the source code of the valid target enclave for compliance against the same privacy policy as the one displayed to the user.

We note that this function can be called multiple times with replays of previous ciphertext data that was sent by the client. In other words, as the TLS connection can terminate outside the target enclave in our threat model and the adversarial OS can replay a client's old public key and ciphertext data to the target enclave. The target enclave would simply encrypt this data to the decryptor enclave, resulting in this function being invoked. Of course, as within our current design the decryptor enclave always decrypts all data that is sent to it, following a successful authentication of the target enclave, the target enclave would have already obtained the plaintext client data. It would simply obtain the same plaintext data again and as it was signed by a valid verifier enclave, it would handle the data in a privacy-compliant way. We remark that we do not protect against adversaries whose aim is not to obtain plaintext data but to repeat some privacy-violating stateful action, such as sending spam email.[1] Therefore, we have shown the security of the decryptor enclave stands against an adversarial OS that attempts to call the enclave's ecalls in an arbitrary order and an arbitrary number of times.

[1]To protect against this attack, the decryptor enclave needs to maintain a list of hashes of values of the $client\_public\_key$ and $ciphertext\_form\_data$ variables in Algorithm 6 and reject any incoming values of these variables that are the same as the old ones.

5.2 Adapting the Graphene-SGX platform

It is evident that local and remote attestation and sealing are essential in our design: specifically, we require the verifier and the target enclaves to be able to attest to the decryptor enclave. We also need to seal the long-term signing key of the decryptor enclave to disk. If sealing is not available on the trusted hardware platform, then the client would need to perform remote attestation with the decryptor enclave *each time* that it wishes to access a website.[2] However, as we described in Chapter 2, Graphene-SGX, in its current form, does not support local or remote attestation. It also does not support sealing or unsealing of content to disk.

We enable Graphene-SGX to support local attestation with a native Intel SGX SDK application as follows. We include stripped-down versions of the native Intel SGX SDK libraries for attestation-relevant functions within the target application and verifier's Graphene-SGX manifest as trusted files. Attestation-relevant functions are not modified themselves, but other functions that are called by these functions are tuned to run on Graphene-SGX; that is, without references to other SGX SDK libraries that are used within the native setup but not within Graphene-SGX. Similarly, we support sealing for the verifier enclave on the Graphene-SGX platform by simply linking the verifier executable against a modified version of the appropriate SGX library and including it as a trusted file within the verifier's manifest. To enable passing local attestation messages between Graphene-SGX enclaves and native Intel SGX enclaves, we design Google Protocol Buffers [Goo19] classes for these messages.

Within the native Intel SGX SDK setup, the untrusted application makes ecalls to the initiating enclave, which then makes ocalls to an untrusted local attestation core object within the untrusted binary. This core object then makes ecalls to the responding enclave. However, in our use case, we have different applications, namely the verifier and the target enclaves, initaiting local attestation to the decryptor. Therefore, we modify the Intel SGX SDK application for local attestation to support this use case and to support Google Protocol Buffers for exchanging messages.

5.3 Verifier enclave

The verifier enclave is responsible for verifying that the PHP source code files, which are to be run within the target enclave, are compliant with a model of the privacy policy. Secondly, in case the source code is found to be compliant with the privacy policy model, the verifier is

[2]The verifier enclave can be functional on a platform that does not support sealing: it would simply regenerate a new keypair to sign the target enclave each time and inform the decryptor enclave of the same.

Algorithm 7 Main algorithm followed by the verifier enclave

1: **function** VERIFIER_MAIN($target\_enclave\_manifest\_path$,
$source\_code\_files\_paths$,
$privacy\_policy\_file\_path$,
$signature\_token\_path$
$verifier\_keypair\_path$) ▷ This function is run within a Graphene-SGX enclave.
2: $(analysis\_result, hashes) \leftarrow source\_code\_analysis(source\_code\_files\_paths,$
$privacy\_policy\_model\_path)$
3: **if** $analysis\_result == False$ **then**
4: **return** 0
5: **end if**
6: ▽ Initializes the signing, verification keypair variables from a sealed keypair on disk.
7: $set\_signing\_keypair(verifier\_keypair\_path)$
8: $complete\_and\_sign\_manifest$ ($target\_enclave\_manifest\_file\_path$,
$source\_code\_files\_paths$,
$hashes, signing\_key, signature\_token\_path$)
9: $target\_mr\_signer \leftarrow compute\_sha256\_hash(verification\_key)$
10: $decryptor\_session\_key \leftarrow do\_local\_attestation(decryptor\_ipc\_channel)$
11: $ciphertext \leftarrow encrypt(decryptor\_session\_key, target\_mr\_signer)$
12: $send\_message(decryptor\_ipc\_channel, ciphertext)$
13: **end function**

also responsible for signing the target enclave, using a long-term signing-verification keypair that is sealed to the disk. Finally, the verifier enclave should conduct local attestation with the decryptor enclave and over the resulting secure channel, inform it of a hash of the verification key, $\mathcal{P} = Hash(\mathcal{VK}_V)$ and of a hash of the privacy policy against which the source code was verified $Hash(PP)$.

The verifier enclave should take as inputs the source code files that are to be run by the target enclave at runtime, and any other inputs that it needs to sign over to generate the target enclave's signature token. As the target enclave is run within Graphene-SGX, the target enclave's manifest file is also included within the signature. Therefore, as input arguments, the verifier enclave takes in the paths to the target enclave's manifest file, its PHP source code files, including the privacy policy model file, and the path where the signature token should be written to disk. We begin with a discussion of the main algorithm that the verifier follows, which is also depicted in Algorithm 7.

The verifier enclave first initializes its long-term signing verification keypair by calling the wrapper function $set\_signing\_keypair()$ which was defined in Algorithm 1. It then runs the

main source code analysis function with the path to the source code files and the privacy policy model file. After verifying the source code files for compliance with the privacy policy model file, this function returns a binary verification result and a list of hashes of all these files. We discuss later on in this section so as to why we modify the source code analysis function to do this. Following a successful verification result, the verifier obtains a signing-verification keypair (line 5), through an implementation of the $get\_signing\_keypair()$ algorithm, which is outlined in Section 4.2.1. It then passes the signing key, the target enclave manifest file path, the hashes that it obtained from the source code analysis tool, and the signature token path to the manifest-signing Python script (line 6). This script has been modified to take into account the fact that it generates the signature token for the target enclave, while being executed on an *untrusted* website provider host machine, as we explain below. It writes the completed manifest file and the signature token to disk.

After writing the signature token, the verifier computes a hash of the verification key (line 7). It then conducts local attestation with the decryptor enclave on a predefined inter-process communication (IPC) channel (line 8). It does not authenticate the peer enclave: that is, it does not check the enclave measurement or the signer measurement of the decryptor enclave. On the secure channel established with the peer enclave through local attestation, the verifier enclave sends the hash of the verification key along with a hash of the privacy policy model file (lines 9, 10). In other words, it encrypts these hashes using its symmetric session key that is shared with the peer enclave, and sends them over the untrusted IPC channel.

Before we describe the $source\_code\_analysis$ and the $complete\_and\_sign\_manifest$ functions, it is necessary to understand the configuration of the verifier's manifest file, which is prepared on a trusted machine. As we mentioned earlier, within Graphene-SGX, files can only be set as either *trusted* files, implying that their hash is included within the manifest and it affects the enclave measurement, or as *allowed* files, so that their hash does not affect the enclave measurement *and* the content of these files could change between successive reads. First, the verifier's manifest should allow the verifier enclave to read its input arguments, namely the target enclave's manifest template file, its PHP source code files and the privacy policy file. It is evident that the PHP source code files cannot be set as trusted files in the verifier's manifest: we do not wish the enclave measurement of the verifier enclave to be dependent on that of the files that it verifies. We therefore set the source code files as allowed files within the verifier's manifest. Secondly, any libraries required to run the two aforementioned functions should be included as trusted files within the manifest. This ensures that the verifier only executes code, through any dynamic libraries that are loaded, whose integrity is checked at runtime.

Algorithm 8 Algorithm for Graphene-SGX's modified manifest-completion and signing script

global variables
  $target\_executable\_hash$
  $dependency\_hashes$ ▷ Hashes of all dependencies of target executable
  $graphene\_library\_hash$
  $target\_executable\_info$ ▷ Information about the target binary in order to compute the enclave measurement and generate a signature
  $graphene\_library\_info$
end global variables

▽ This function is run within a Graphene-SGX enclave.
function COMPLETE_AND_SIGN_MANIFEST($target\_enclave\_manifest\_file\_path$, $source\_code\_files\_paths$ $source\_code\_privacy\_policy\_model\_hashes$, $signing\_key$, $signature\_token\_path$)
  $target\_manifest\_file \leftarrow file\_open(target\_enclave\_manifest\_path)$
  $(allowed\_files\_list, trusted\_files\_list) = parse\_file(target\_manifest\_file)$
  if $allowed\_files\_list \neq Empty$ **then**
    return 0
  end if
  if $source\_code\_files\_paths \not\subset trusted\_files\_list$ **then**
    return 0
  end if
  $append\_to\_file(target\_manifest\_file,$ $source\_code\_privacy\_policy\_model\_hashes,$ $target\_executable\_hash, dependency\_hashes,$ $graphene\_library\_hash)$
  $signing\_material \leftarrow compute\_signing\_material(target\_manifest\_file,$ $target\_executable\_info,$ $graphene\_library\_info)$
  $signature\_token \leftarrow sign(signing\_key, signing\_material)$
  $write\_to\_disk(signature\_token, signature\_token\_path)$
end function

complete_and_sign_manifest: The main algorithm followed by the manifest completion and signing script is shown in Algorithm 8. This algorithm is passed the path of the target enclave's manifest file, hashes of the source code and privacy policy files, the long-term signing

key and the disk path where the signature token should be written to. This function first opens and parses the manifest template file (line 9,10) to extract the set of paths of *trusted* and *allowed* files within the manifest. In order to stop arbitrary code from being executed in the target enclave, this algorithm ensures that the set of *allowed* files is empty (lines 11–12). In other words, the target enclave manifest file should only contain *trusted* files. Importantly, we note that this check ensures that the integrity of the PHP source code files run within the target enclave is checked at runtime.[3]

If the target enclave's manifest file satisfies this check, then the PHP source code files could either be set as *trusted* files or excluded entirely from the manifest template. In the latter case, Graphene-SGX will not allow the target web server enclave to load its source code files at runtime, leading to a denial of service attack. To detect if developers have erroneously left out setting PHP source code files as *trusted* ones in the manifest template, this function reports an error and halts if the list of paths of all trusted files does not contain the path of the source code files, which is passed as an input argument (lines 13–14). After performing these two checks over the content of the target enclave manifest, this function appends hashes of the source code files, which were also passed as input arguments to the manifest file (line 15).

As mentioned previously, complete Graphene-SGX manifests include hashes of the target executable and all libraries that the target executable depends on. Within our threat model, the adversarial host OS can provide any view of the filesystem as it desires, and thus, it can provide malicious, privacy-invasive versions of the above files to any system calls run within this function. As the verifier enclave does not verify the source code for the target executable or for its dependencies, this script contains hard-coded values for the above hashes (lines 2–3). It appends these hashes to the manifest file (line 15). Now, Graphene-SGX enclaves consist of the enclave executable, a Graphene-SGX library and the enclave manifest. Consequently, the computation of the enclave measurement for Graphene-SGX enclaves requires information about these files. As the adversarial OS can provide inaccurate information about the enclave executable or the Graphene-SGX library for the computation of the enclave measurement, this information is also hard-coded into the script and used to compute the signing material (line 16). Finally, the script signs the signing material, which includes the enclave measurement and other enclave attributes, with the signing key that was passed to it as input (line 17) and writes the file to disk.

[3]If the PHP source code files are set as *allowed* files, then an adversarial operating system can perform a TOCTTOU attack that breaks the soundness guarantees of Mitigator. That is, the adversarial OS can easily replace a verified version of the files that is signed by the verifier with a malicious version when the target enclave is being run.

Algorithm 9 Algorithm to use source code analysis tool

```
1: function SOURCE_CODE_ANALYSIS(source_code_files_paths, privacy_policy_file_path)
   ▷ This function is run within a Graphene-SGX enclave.
2:     source_code_files ← file_open(source_code_files_paths)
3:     source_code_files_content ← file_read(source_code_files)
4:     privacy_policy_file ← file_open(privacy_policy_file_path)
5:     privacy_policy_file_content ← file_read(privacy_policy_files)
6:     hashes ← compute_hashes(source_code_files_content,
                                privacy_policy_file_content)
7:     privacy_policy_model ← form_privacy_policy_model(privacy_policy_file_content)
   ▷ The following function performs the compliance check on the opened files.
8:     verification_result ← compliance_check(source_code_files_content,
                                              privacy_policy_model)
9:     return(verification_result, hashes)
10: end function
```

source_code_analysis **:** We illustrate the main algorithm followed by Pixy in Algorithm 9. Note that the source code analysis function has been modified to compute hashes over the source code and the privacy policy files, through the $compute\_sha256\_hashes$ function (line 4). The reasoning behind this decision follows. We have seen that the $complete\_and\_sign\_manifest$ function includes hashes of the source code files and the privacy policy file within the manifest. Furthermore, as we explained above, these files can only be set as *allowed* files within the verifier manifest, and thus their integrity will not be checked when the verifier enclave opens them at runtime. Next, it is evident that all of these files must be opened in order to perform source code analysis. Opening and performing computations on these files twice, without checking their integrity, leaves the OS with an opportunity to perform a TOCTTOU attack by providing a compliant copy of the files to the source code analysis function and a non-compliant copy to the $complete\_and\_sign\_manifest$ function. Thereby, the OS obtains a signature over non-compliant source code files and in particular, this attack implies that even if the original source code analysis function $compliance\_check$ is sound, the verifier enclave would not be sound. In order to prevent this attack, we first store the content of the file in enclave memory (lines 3,5), compute hashes over this content (line 7) and return these hashes to the $verifier\_main$ function (line 9), which then passes them to the $complete\_and\_sign\_manifest$ function for inclusion in the manifest. We use the content of the privacy policy file in enclave memory to form the privacy policy model (line 6). Currently, this function is an identity function but it could use sophisticated natural language processing tools, such as those outlined in Section 2.1, to derive the privacy policy model. The privacy policy model and the in-enclave

source code files' content is input to the $compliance\_check$ function, which checks the compliance of the source code (line 8). We remark that we thus always refer to the same copy of the files throughout the $source\_code\_analysis$ function. Given that we have described how the $compliance\_check$ function is *used* within the verifier enclave, we can proceed to discuss how it is implemented using the Pixy taint analysis tool.

5.4 Adapting a source code analysis tool: Pixy

We described Pixy's functionality as a PHP source code analysis tool in Section 2.2. We repeat that Pixy uses input arrays to PHP scripts, such as POST or COOKIE arrays, as taint sources. It also includes a configurable blacklist of output functions that it regards as taint sinks and another configurable whitelist of functions that it regards as sanitization functions, which take in tainted input and produce untainted output. We use Pixy within the verifier enclave to analyze the source code of the target enclave's PHP files. We recognize that our privacy compliance problem can be phrased as a taint analysis problem as follows:

1. Incoming POST array values are tainted. However, these may be directly logged or shared, as they are encrypted to the decryptor enclave. Marking code that does this as vulnerable would then only increase the false positive rate of Mitigator. That is, it would affect the completeness of the privacy compliance checking but not its soundness.

2. If a tainted incoming POST array value is passed into the target enclave's method for decrypting data, then the output of that function must also be tainted. That is, this function should not be treated as a sanitization function. This is essential as otherwise, flows of the plaintext data present in the output will no longer be tracked by Pixy and consequently, the soundness of the privacy compliance checking is affected. (We remark that in an ideal case, our source code analysis tool would not taint incoming POST array values but simply taint the plaintext data that is output by this function. However, Pixy does not easily support a configuration of function outputs as taint sources.)

3. The blacklist of output functions that is considered as taint sinks must include all functions that the current privacy policy disallows. A blacklist that misses such functions will affect Mitigator's soundness.

We implement restrictions 2 and 3 above, using Pixy. We leave condition 1 for future work, as it only affects the completeness but not the soundness of Mitigator. We observed that by default, for functions whose definitions are not in the source code file, Pixy marks the outputs of

such functions as untainted, even if the inputs are tainted. That is, Pixy treats these functions as not propagating taints from input parameters to the returned value. Mitigator's target enclave's method for decrypting client-side data, which is abstracted as the $php\_decrypt\_wrapper$ method in Algorithm 10 is treated this way.[4] We therefore modify Pixy to treat such functions as potentially dangerous, that is, as if they *do* propagate taints from any input to the output. Note that this is a more versatile option that simply hard-coding the specific function as a taint-propagating one.

Currently, Pixy is designed to mark the same set of functions as output functions for inputs coming from all different arrays (GET, POST, COOKIE, etc.). It can easily be customized to implement a more granular policy for different arrays, such as the POST or the COOKIE array, as well as for different elements in each array, which correspond to names of elements on form fields for POST arrays and cookie names for COOKIE arrays. Finally, we observe that if Pixy marks the target enclave source code as being non-compliant with the privacy policy model, then developers should modify the source code so that it is compliant with the privacy policy model. Alternatively, the privacy policy can be modified to reflect what the organization actually does with the data, in terms of the source code. In case Pixy (as it is sound but incomplete) wrongly marks compliant source code as non-compliant, the developers may simplify the source code so that Pixy correctly perceives the source code as being compliant.

5.5 Target enclave—PHP Extension

Graphene-SGX allows us to run a native, unmodified web server within an Intel SGX enclave. We use Graphene-SGX to run an Apache server that hosts PHP source code files. Mitigator's PHP extension acts on behalf of the target enclave and implements its functionalities outlined in Section 4.2. It is responsible for three tasks. First of all, it should perform local attestation with the decryptor enclave once when the server is set up. From the local attestation shared secret, it should derive the key k_{DT}. Secondly, after the PHP extension has performed local attestation, any PHP scripts should be able to call into the extension to obtain the token H that is to be sent with every HTTP response. Finally, in Mitigator, the browser extension would encrypt form field data to the decryptor enclave and send in the client's public key pk_C. The PHP extension should enable any calling PHP scripts to pass this key, the client's ciphertext data and expect plaintext client form fields in response. Mitigator's PHP-CPP extension provides

[4] Any methods linked at runtime in the PHP extension are treated as not propagating taints. As mentioned in Section 5.5, a PHP extension is used to implement the target enclave's functionalities. The $php\_decrypt\_wrapper$ method is linked at runtime to its definition in the PHP extension and is thus treated this way.

an implementation of the interface shown in Algorithm 10, in order to perform the abovementioned three tasks. Specifically, it contains the functions $local\_attestation\_and\_first\_message$, $get\_mitigator\_public\_key\_token$, and $php\_decrypt\_wrapper$, which we briefly outline below.

local_attestation_and_first_message: This function performs the first aforementioned task: it initiates a local attestation request to the decryptor enclave and conducts the rest of the local attestation as the initiating enclave. The target enclave does not need to authenticate the decryptor enclave and so the extension does not check the enclave and signer measurements of the responding enclave. Following local attestation, the function derives a symmetric key for future communication with the decryptor enclave. Additionally, it waits for the decryptor enclave to send an encrypted value for the token H. It decrypts the first message from the decryptor enclave and stores the resulting plaintext.

get_mitigator_public_key_token: After the above function is executed once at PHP startup time, any PHP scripts can call the $get\_mitigator\_public\_key\_token$ function to retrieve this token.[5] The calling PHP script may then set the value of a custom header, known as the `Mitigator-Public-Key` header, to that of the token H in HTTP responses to requests for pages with forms.

php_decrypt_wrapper: PHP scripts on pages that obtain form data from the client may similarly expect to obtain the client's public key from the value of another custom header, say `Mitigator-Client-Public-Key`. To process encrypted form fields sent by the browser extension, such PHP scripts can pass the value of this header, which is the client's short-term public key pk_C, and any ciphertext form fields to the function $php\_decrypt\_wrapper$. Internally, the function encrypts the ciphertext form field values to the decryptor enclave, using the stored symmetric key k_{DC} and then sends them over an untrusted IPC channel to the decryptor enclave. It proceeds to wait for a response from the decryptor enclave. On getting a response, the decryptor enclave decrypts it using k_{DC} and then returns the plaintext form fields to the invoking PHP script.[6]

We note that exchanging the short-term public keys could be done through means other than HTTP headers: the PHP extension can publish the token H on a designated page on its website,

[5]The reader may observe that the above pseudocode for the $get\_mitigator\_public\_key\_token$ function returns the same token throughout the lifetime of the decryptor enclave. In particular, it effectively returns the long-term public key for the decryptor enclave. As long as the client-side extension generates a truly ephemeral keypair, the shared secret k_{DC} will differ for each client and the OS would not be able to decrypt the client's ciphertext without passing it to the decryptor. This function can be modified to obtain a new token each time by communicating with the decryptor application and signalling it to call the ecall $ecall\_create\_and\_encrypt\_mitigator\_token$, which returns a new token.

[6]The current implementation of these functions is not thread-safe. Graphene-SGX does not support shared memory at the time of this writing and due to time constraints we did not implement signal-based mechanisms for resource sharing to make these functions thread-safe.

from which the browser extension can be expected to retrieve the token. The browser extension may also support setting up another secure channel to send its public key pk_C. A main advantage of using HTTP headers for communicating Mitigator-specific values between the client and Mitigator's server-side PHP extension is that retrieving and setting these headers is an integral feature of all major server-side languages, such as PHP and Javascript. Additionally, Mitigator-specific tokens piggyback on the same secure TLS channel used to receive and send these values as the webpage itself, therefore requiring no additional communication overhead to set up a secure channel. Moreover, no additional round trips to the server are incurred to retrieve these values.

Algorithm 10 The PHP extension implements these three functions and maintains state corresponding to the global variables below. All of these functions are run within Graphene-SGX enclaves.

global variables
  $decryptor\_session\_key$
  $mitigator\_server\_header$
end global variables

function LOCAL_ATTESTATION_AND_FIRST_MESSAGE() ▷ This function conducts local attestation and sets the two global variables.
  $do\_local\_attestation\_as\_initiator(decryptor\_ipc\_channel)$
  $ciphertext\_mitigator\_header \leftarrow receive\_message(decryptor\_ipc\_channel)$
  $(decrypt\_status, message) \leftarrow decrypt(decryptor\_session\_key, ciphertext\_mitigator\_header)$
  if $decrypt\_status \neq False$ **then**
    return "Error in decrypting: OS might have modified the token"
  end if
  $mitigator\_server\_header \leftarrow message$
end function

function GET_MITIGATOR_PUBLIC_KEY_TOKEN()
  return $mitigator\_server\_header$
end function

function PHP_DECRYPT_WRAPPER($client\_public\_key, client\_ciphertext$) ▷ This function obtains client's plaintext data, by communicating with the decryptor enclave.
  $ciphertext\_to\_decryptor \leftarrow encrypt(decryptor\_session\_key, client\_public\_key \| client\_ciphertext)$
  $send\_message(decryptor\_ipc\_channel, ciphertext\_to\_decryptor)$
  $ciphertext\_from\_decryptor \leftarrow receive\_message(decryptor\_ipc\_channel)$
  $plaintext\_client\_data \leftarrow decrypt(decryptor\_session\_key, ciphertext\_from\_decryptor)$
  return $plaintext\_client\_data$
end function

Algorithm 11 The browser extension maintains state corresponding to the variables below. It has the following interface of functions to encrypt form field data to the decryptor enclave and to set the client's header before sending HTTP submissions of forms.

global variables
    $valid\_verifier\_enclave\_measurements \leftarrow \mathcal{L}_1$
    $url\_verification\_key\_map \leftarrow \mathcal{L}_2$
    url_key_map
end global variables
function ENCRYPT_FORM_FIELD(plaintext_form_field) ▷ This function is called after the user presses a button to encrypt its form field data.
    $decryptor\_session\_key \leftarrow url\_key\_map[url]$["derived"]
    $ciphertext\_form\_data \leftarrow encrypt(decryptor\_session\_key, plaintext\_form\_field)$
    return $ciphertext\_form\_field$
end function

function SEND_OWN_PUBLIC_KEY_HEADER(url) ▷ This function is called to set the custom `Mitigator-Client-Public-Key` header before sending the headers for a request from a page with forms.
    $own\_public\_key \leftarrow url\_key\_map[url]$["own_public"]
    return "Mitigator-Client-Public-Key:own_public_key"
end function

5.6 Browser extension

In this section, we provide an overview of how the browser extension interacts with the target enclave and how it securely obtains users' inputs. First, the browser extension is responsible for sending and receiving Mitigator-specific tokens to and from the PHP extension. As mentioned in Section 5.5, custom HTTP headers are used to communicate public key-related information between the decryptor enclave and the browser extension. Specifically, to send the decryptor's token H to the client, we use the `Mitigator-Public-Key` header. In the other direction, the `Mitigator-Client-Public-Key` header is used to send the client's public key to the decryptor enclave. The latter header is set and sent through the $send\_own\_public\_key$ function in Algorithm 11.

Algorithm 12 The browser extension has the following function to verify the decryptor enclave's header, to generate its own keypair and to derive a shared secret for future encryption of form field data.

▽ This function is called if and as soon as the browser extension receives the `Mitigator-Public-Key` header.
function VERIFY_MITIGATOR_PUBLIC_KEY_HEADER_DERIVE_KEY(url, header_value)
    $(token, signature) \leftarrow header\_value$
    $verification\_key \leftarrow url\_verification\_key\_map[url]$
    $signature\_valid \leftarrow verify\_signature(verification\_key, signature, token)$
    if $signature\_valid == False$ **then**
        return False ▷ Token was signed by an impostor decryptor enclave or the untrusted OS.
    end if
    $(decryptor\_public\_key, verifier\_mrenclave, privacy\_policy\_hash) \leftarrow token$
    if $verifier\_mrenclave \notin valid\_verifier\_enclave\_measurements$ **then**
        return False ▷ An invalid verifier enclave was used to conduct the compliance check.
    end if
    $displayed\_privacy\_policy\_hash \leftarrow compute\_privacy\_policy\_hash(url)$
    if $displayed\_privacy\_policy\_hash \neq privacy\_policy\_hash$ **then**
        return False ▷ Compliance was not checked against the displayed privacy policy.
    end if
    $url\_key\_map[url]$["decryptor_public"] $\leftarrow decryptor\_public\_key$
    $(own\_private\_key, own\_public\_key) \leftarrow create\_keypair()$
    $url\_key\_map[url]$["own_public"] $\leftarrow own\_public\_key$
    $decryptor\_session\_key \leftarrow derive\_key(own\_private\_key, decryptor\_public\_key)$
    $url\_key\_map[url]$["derived"] $\leftarrow decryptor\_session\_key$
    return True ▷ Success
end function

Second, the browser extension should verify the integrity of the server-side Mitigator token that it receives. It does so by performing the three checks outlined in Section 4.2.3 over the token. Mitigator's browser extension currently contains hard-coded values for the lists of valid enclave measurements ($\mathcal{L}_1$) and of verification keys corresponding to a decryptor enclave behind each Mitigator-supporting website ($\mathcal{L}_2$) and it maintains them in its state corresponding to Algorithm 11. (We discussed in Section 4.2.4 that owing to high network and computational costs to establish these lists in a trustworthy manner, a user may reasonably delegate a privacy advocacy

organization to populate these lists and make them publicly accessible on their website.)[7] We illustrate these series of checks in the function $verify\_mitigator\_public\_key\_header\_derive\_key$ function in Algorithm 12. If any of these checks fail, then the browser extension proceeds as if the website does not support Mitigator. In other words, it does not derive a symmetric key to encrypt its data to the decryptor enclave, as the authenticity of the Mitigator infrastructure on the server-side machines cannot be verified. Additionally, the user interface is not changed to reflect that Mitigator is supported unless all checks succeed. However, if the checks succeed, then after Mitigator's browser extension derives the symmetric key, its icon turns to a different colour, to signal the user to press on it.

Finally, the browser extension should encrypt form field data to the server-side script and send the ciphertext instead of plaintext form field data. Currently, the user simply enters their form field data into the webpage itself. When the user finishes typing their text, they can press a submit button that is at the bottom of this pop-up window, which results in the entered text fields to be encrypted to the decryptor enclave for the given website. We encapsulate this through the $encrypt\_form\_fields$ function in Algorithm 11.

We remark that the current implementation can be subject to user interface attacks similar to those pointed out by Freyberger et al. [FHA+18]. With some changes to the browser extension, a design motivated from the final design discussed by Krawcieka et al. in SafeKeeper [KKP+18] can be implemented, as follows.[8] The user is expected to signal Mitigator's browser extension into a mode to securely enter their data by pressing on the Mitigator icon. When the user presses on this icon, it produces a pop-up that replicates the functionality of the form but not its appearance. Specifically, it contains a plain HTML form with the same number of text fields as in the original form, in the same order and with the same labels, but does not contain any Javascripts from the original webpage. Again, as the user presses the submit button, the entered text fields are encrypted and sent to the website.

[7] We remark that Krawiecka et al. [KKP+18] present a zero round-trip remote attestation protocol for the case wherein only one enclave needs to authenticate to the other. For our case, this becomes relevant as only the decryptor enclave, behind a given Mitigator-supporting URL, needs to authenticate itself to the privacy advocacy organization's client or enclave. In other words, the latter does not need to authenticate itself to the decryptor enclave as we expect any party to be able to verify the decryptor enclave's identity. The privacy advocacy organization's client can include an implementation of SafeKeeper's remote attestation protocol [Kur18] to efficiently verify the integrity of Mitigator's decryptor enclaves.

[8] We could not implement these changes on account of time constraints.

5.7 Summary

In this chapter, we have outlined the implementation of our design. In the verification stage, the website provider runs the verifier enclave on Graphene-SGX with the target enclave source code files and the target enclave's Graphene-SGX manifest template as inputs. The verifier enclave checks the compliance of the source code against the privacy policy file, which is a simple configuration file for the Pixy source code analysis tool. We assume soundness of the source code analysis tool; that is, if the source code is non-compliant then the tool will report this non-compliance. We carefully design the verifier enclave such that given this assumption, it cannot sign a non-compliant target enclave.

The verifier enclave then proceeds to the post-verification stage, wherein it performs local attestation with the decryptor enclave, and importantly, it sends authentication information for the target enclave that it has signed. At runtime, the target enclave handles messages from the decryptor enclave and the client's data from the browser extension through a PHP extension. When started up, the target enclave performs local attestation with the decryptor enclave, and the decryptor enclave sends it a token that it can then send to clients as a header. Mitigator's browser extension verifies the integrity of this token, encrypts data to the decryptor enclave using its public key from the token and sends its own token as another header. This browser extension is also responsible for performing remote attestation with the decryptor enclave to ensure its authenticity and the client may delegate this task to a trusted privacy advocate organization. We design the ecalls of the decryptor enclave such that given that the browser extension checks the integrity of the token and the verifier enclave is sound, the client's data will only be passed to a compliant target enclave.

Chapter 6

Performance Evaluation

As Goldberg [Gol07] remarks, for privacy-enhancing technologies to impact people's lives, users should *want* to use those technologies and observes that, for instance, users may turn off or stop using systems that take longer to run than their non-privacy-enhancing counterparts. In order to guage whether different users want to use Mitigator, a detailed user study is required. We leave such a study to future work. Instead, we focus on the second aforementioned point: for Mitigator to gain widespread adoption, it is desirable that it should *not* incur a large overhead in comparison to an identical system without Mitigator, so as to not dissuade users through a time overhead. We conduct a set of experiments to measure the latency of a website that runs Mitigator at runtime in comparison to a website run without Intel SGX. We remark that the verification stage incurs a small one-time latency overhead during the development process, and thus do not test it further as it does not impact the runtime latency experienced by users. We also compare the latency of a website with Mitigator to the latency of one run within Graphene-SGX but without Mitigator. We highlight that the purpose of these experiments is to identify and explain any sources of significant latency in comparison to the above systems, rather than to *optimize* Mitigator to decrease its overall runtime latency.

We begin with a description of how our server-side and client-side components were set up in order to conduct experiments in Section 6.1. We proceed with a description of how we instrument these components in order to measure timestamps or time durations in Section 6.2. We measure the latency of our server-side components through a series of experiments that we describe in Section 6.3. Finally, we discuss experiments to measure the end-to-end latency of our system in Section 6.4.

6.1 Setup

We set up our server-side system on a four-core Intel i5-6500 CPU that supports Intel SGX. The machine runs Ubuntu 16.04.4. We set up and installed the Intel SGX SDK and Graphene-SGX on this machine. We ran the verifier enclave on valid target enclave source code and a sample configuration file as the privacy policy file and it output a signed target enclave. The target enclave runs an Apache server executable within Graphene-SGX which links to the Mitigator PHP extension and serves three PHP files. This Apache server is known as the *Mitigator server*. Once the target enclave has been signed, we run the decryptor and target enclaves through the post-verification stage. Following the local attestations outlined in Figure 4.3, the target enclave is ready to serve the PHP files.

The first PHP file displays a form and a link to the privacy policy file. Responses to HTTP requests for this page include our custom `Mitigator-Public-Key` header whose value is set to the token H. Upon receiving this HTML form page, the browser extension verifies the value of this header. As we discuss below, for these performance experiments, we modified the browser extension to automatically generate plaintext form field values that consist of random plaintext characters, to encrypt these form fields, and send the resulting ciphertext form fields to the second PHP file through an HTTP POST request, or to log them, in order to be sent later. It also sends the value of the `Mitigator-Client-Public-Key` header in the same request, or logs it, in order to enable the decryptor enclave to decrypt the ciphertext form fields. This second PHP file passes the values of any POST array variables that it received, along with the value of the `Mitigator-Client-Public-Key` header, to an implementation of the $php\_decrypt\_wrapper$ function, which is present in the PHP extension and was discussed in Section 5.5, in order to obtain the plaintext form fields. The second PHP script then simply prints the output of this function, namely, the plaintext form fields. We note that the browser extension can send a variable number of form fields with data of variable length to the second PHP script in an HTTP POST request as all of these POST array values are passed to the above function.

Apart from running the target enclave, we also run another Apache server within Graphene-SGX that does not load the PHP extension and differs in the second PHP file. This file just prints the values of any POST array variables that it receives. We refer to this server as the *Graphene-SGX* server. Finally, we run another Apache server outside of the SGX platform that does not load the PHP extension, and we refer to it as the *control* server. We remark that we do not set up any of the aforementioned three servers with TLS support, owing to time constraints; however, we believe that our experiments can be replicated on servers with TLS support.

We now proceed to explain the modifications to the browser extension to conduct our tests. First, we modified the browser extension to automatically generate readable plaintext characters

as form field data and to encrypt each such form field value to the decryptor enclave. We support generating requests with a variable number of ciphertext form field values or with variable size of a single form field, as required for our experiments. (We remark that as this is a preliminary evaluation, we only test the change in latencies with textual form fields; we leave the measurement of latencies for encrypting other data types to future work.) Therefore, for instance, we can generate n requests for each value of the control variable; that is, we send n requests for a given number of form fields or for each form field size. We remark that we first send a single request for each of the values of the control variable (form field length or number of form fields) and then send another request and so on, until n requests are sent for each value of the control variable. This is done to prevent the network lag or other variables from affecting the observed latencies for one value of the control variable more than others. For the same reason, we send each such request to each of the servers consecutively before sending another request with a different value of the control variable rather than sending *all* requests first to the Mitigator server, then to the Graphene server and then to the plain Apache server. Finally, in order to emulate sending such requests from a client on the same network as the servers, wherein the client is headless, we also modify the browser extension to log the ciphertext form fields, the expected plaintext form field outputs and the value of the `Mitigator-Client-Public-Key` header.

6.2 Instrumentation

We instrument the servers, the server-side enclaves, and the browser extension to measure the wall clock time that has passed during the execution of relevant functions, as follows. We instrument the servers by simply obtaining timestamps at the start and at the end of the PHP scripts for the second PHP page for each of the servers. The difference of these timestamps is equal to the wall clock time that has passed in the execution of the PHP script. We instrument the target enclave, that is, the PHP extension, to measure the time that it takes to generate a client's plaintext form field data, given the ciphertext and the client's public key. Specifically, we modify the $php\_decrypt\_wrapper$ function in Algorithm 10 to obtain an initial timestamp before line 18 and a final timestamp after line 21. The time duration defined by the difference of these timestamps will include the time to encrypt the client's ciphertext and public key to the decryptor enclave (line 18), the time for the decryptor enclave to process the message and to generate a ciphertext response, the time to decrypt the decryptor enclave's response ciphertext (line 21) as well as the time for any inter-process communication (lines 19, 20). We also instrument the decryptor enclave to measure the time that it takes to process the target enclave's ciphertext message and return another ciphertext message. That is, we instrument the untrusted decryptor application, which is shown in Algorithm 2, to measure the wall clock time after it receives a message from

the target enclave (line 17) and after it sends a response to it (line 18). This thus includes the wall clock time taken to run the ecall $ecall\_process\_targets\_message$, which internally computes the session key for that client and performs two symmetric-key cipher decryptions and one symmetric-key cipher encryption.

We remark that in our current implementation, the $php\_decryptor\_wrapper$ function will be called once for each form field value that is to be decrypted. Similarly, the aforementioned ecall will be called once for each form field and the shared secret and session key will be recomputed for form fields on the same page and sent by the same client. It is evident that the computation of the shared secret for *each* form field is redundant and the implementation can be optimized to modify the $php\_decryptor\_wrapper$ function to accept all form field values instead of just one, and encrypt all form field values in one ciphertext to the decryptor enclave. Similarly, the ecall can be optimized to compute the shared secret only once and to decrypt all form field values present in the ciphertext from the target enclave. We leave these optimizations for future work but expect that they will result in a significant reduction in the runtime computational overhead of the decryptor and target enclaves, as follows. First, instead of computing the shared secret m times for m form fields, the decryptor enclave would only be computing it once. Similarly, instead of exchanging m ciphertexts between the target and decryptor enclaves for m form fields, these enclaves will only be exchanging one longer ciphertext that contains all m client ciphertext form fields and one copy of the client's public key. Thus, only one encryption and decryption of an m times longer ciphertext will take place instead of m encryptions and decryptions of smaller ciphertexts.

We instrument the browser extension to measure the time that it takes to encrypt a given message; that is, we instrument $encrypt\_form\_field$ function in Algorithm 11, to obtain a timestamp before line 7 and after line 8. We also instrument the browser extension to record a timestamp before sending a request and after obtaining a response, in order to measure the end-to-end network round-trip time.

6.3 Server-side latency

First, we obtain an estimate of the server-side latency of our system by measuring the latency of the Mitigator server's responses to requests from the localhost interface on the servers' machine and contrasting these latencies in comparison to the latencies of responses by the Graphene-SGX and control servers. Sending requests from the localhost interface eliminates the network round-trip time and thereby allows us to precisely measure the server-side latency overhead of using Mitigator. As we also instrument the decryptor and target enclaves, we obtain precise breakdowns so as to how much latency overhead each server-side component contributes.

We conduct two sub-experiments as follows. In the first sub-experiment, we run the browser extension to generate requests with one form field of variable length and to record the expected plaintext responses. The length of the form field in each request falls in the range $32, 64, ...1024$. The second sub-experiment differs from the first one only in that requests with a variable *number* of form fields, such that each form field is of constant length, are generated in the browser extension. The number of form fields in each request falls in the range $1, 2, 4, 8, ...128$. In both sub-experiments, $n = 50$ requests are sent for each value of the independent variable; that is, the number of form fields or the length of a form field. We then send these requests from the localhost interface of the servers' machine to the second PHP page of the Mitigator, Graphene-SGX, and control servers respectively using the *curl* tool. We record the total wall clock time taken to obtain the complete HTTP response, as measured by that tool. The instrumented decryptor and target enclaves output the wall clock time taken within relevant functions run for each of the form fields. The instrumented Mitigator, Graphene-SGX, and Apache server PHP scripts allow us to record the total wall clock time that has passed as the HTML response page was formed.

In Figure 6.1, we plot the averages and standard error bars for the following wall clock times as the length of the single form field increases: the total network round-trip time for each of the three servers, the time spent solely within the PHP script for the Mitigator server, the time spent within the decryptor enclave, and the time spent within the PHP extension. In Figure 6.2 we plot the averages and standard error bars for the same six time measurements for the second sub-experiment; that is, as the number of form fields increases. We remark that the wall clock time measured in the PHP scripts for the Graphene and control servers was 0 in the scale of microseconds and thus we do not plot it on the graphs below.

We draw the following conclusions from Figure 6.1:

1. The latencies and round-trip times are in the order of magnitude of milliseconds and are thus very small in absolute terms. This is because the client is situated on the localhost interface of the server machine. We remark that the increase in the average round-trip time for all three servers for requests with a form field of length 1024 is because the HTTP request is split over two TCP packets instead of just one.

2. The total network round-trip time of the Graphene-SGX server is at most about 2 times as much as that of the control server. We remark that the round-trip time of the control server is shown as 0 ms for all requests with form fields of all lengths other than 1024, as the curl tool measures times to the precision of 1 ms and no less.

3. The decryptor enclave and the PHP extension times each consume at most 10% (0.25 ms out of 2.5 ms) of the Mitigator round-trip time when the client is situated on the localhost interface of the server's machine.

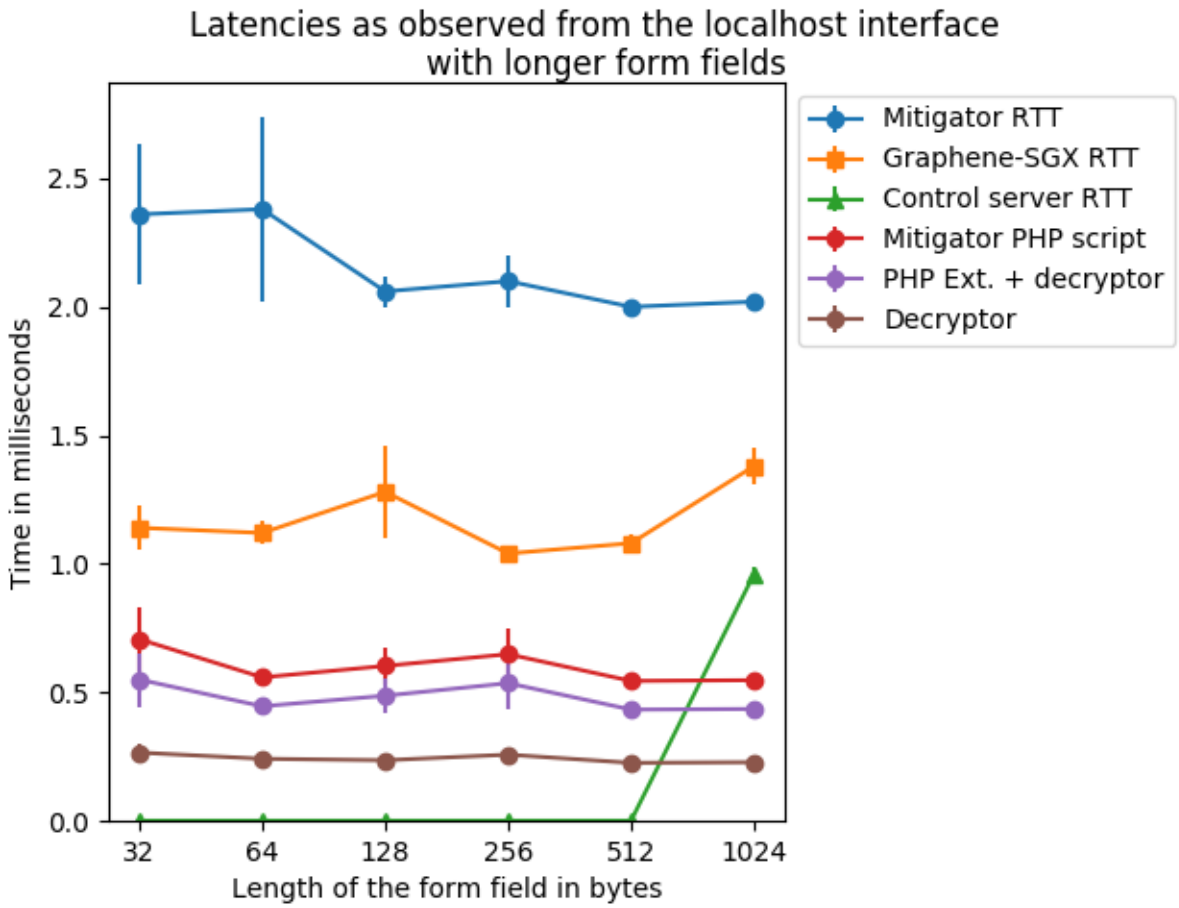

Figure 6.1: Measurement of server-side latency from the localhost interface as the size of a form field increases: the length of a single form field was doubled in steps from 32 to 1024 and we measured the round-trip times (RTT) from the localhost interface for each of the Mitigator, Graphene-SGX, and control servers, and the averages, over 50 requests for each form field length, are plotted in blue, orange, and green respectively. For the Mitigator server, we measured the total wall clock time spent within the PHP script, the total time spent within the PHP extension and the decryptor enclaves and the time spent within the decryptor enclave and the corresponding averages are plotted in red, light purple, and brown respectively.

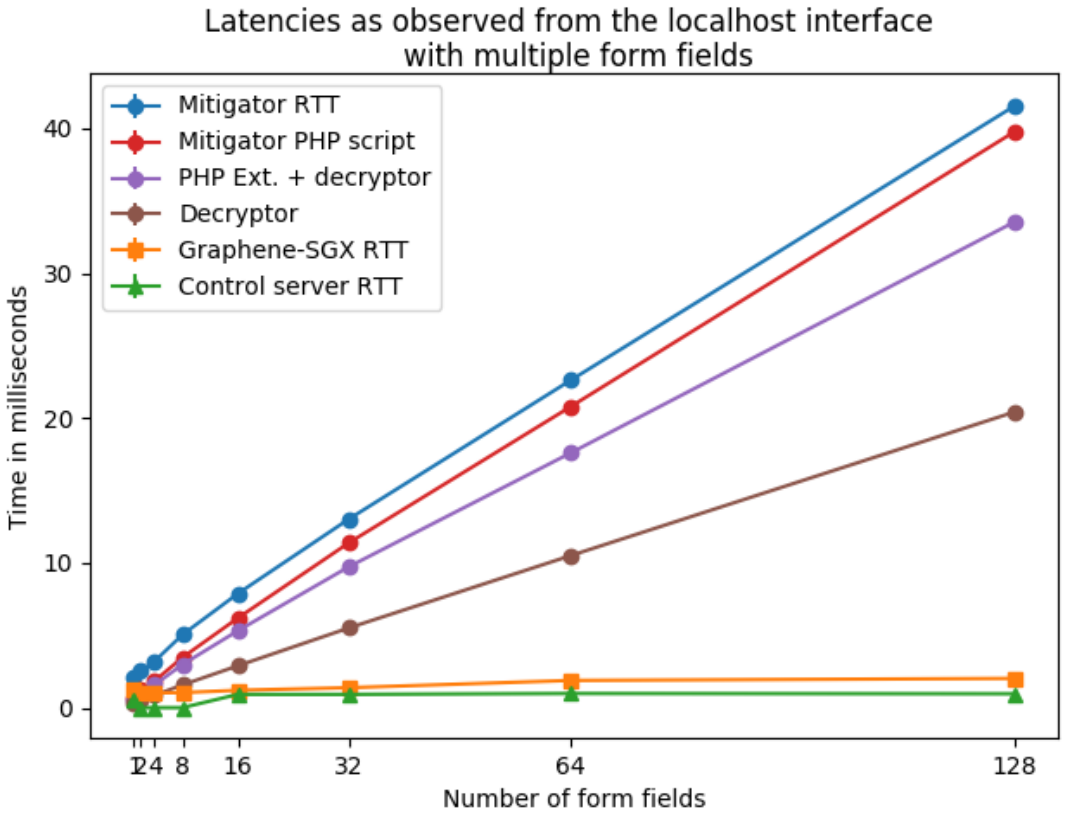

Figure 6.2: Measurement of server-side latencies from the localhost interface as the number of form fields increases: starting from 1, the number of form fields was doubled in steps up to 128, and we measured the round-trip times (RTT) from the localhost interface for each of the Mitigator, Graphene-SGX, and control servers, and the averages, over 50 requests for each number of form fields, are plotted in blue, orange, and green respectively. For the Mitigator server, we measured the total wall clock time spent within the PHP script, the total time spent within the PHP extension and the decryptor enclaves, and the time spent within the decryptor enclave and the corresponding averages are plotted in red, light purple, and brown respectively. The standard error bars for this dataset are too small to be seen on this graph.

4. The decryptor and target enclave times do not change significantly as the size of the form field increases. This is to be expected as both enclaves simply perform symmetric-key cipher operations on longer plaintexts or ciphertexts. In other words, as the length of the client's ciphertext increases, the target enclave encrypts longer ciphertexts to the decryptor enclave. The decryptor enclave then decrypts the longer ciphertext, obtains the client's plaintext and re-encrypts the longer plaintext to the target enclave.

5. The difference in the total network round-trip time for the Mitigator server (blue) and for the Graphene-SGX server (orange) is almost equal to the total computation time spent within the PHP script (red), implying that we have correctly identified the major sources of computational overhead within server-side Mitigator components: namely, the decryptor and target enclaves and the PHP script.

It is evident from Figure 6.2 that observations 2 and 5 above also hold true as the number of form fields in each request is changed. We draw the following observations from Figure 6.2:

6. If multiple form fields are sent, then the total computation time within enclaves (shown in red) constitutes almost all of the total Mitigator round-trip time (shown in blue).

7. In contrast to observation 4 for Figure 6.1 above, the decryptor and target enclave times both increase linearly with the number of form fields. This is to be expected; the decryptor enclave implementation currently involves recomputing the shared secret for each form field, as discussed above. The aforementioned optimizations effectively involves bundling multiple form fields from the client together into a single long ciphertext message at the target enclave and sending this one long message to the decryptor enclave. Thus, we expect that if these optimizations were to be implemented, then the total computation time spent within both enclaves would stay almost constant with an increase in the number of form fields, similar to that observed for the first sub-experiment.

In light of observations 3 and 7 above, we hypothesize that under realistic network conditions, the computational overhead due to Mitigator is very small in comparison to the network overhead. To confirm our hypothesis, we repeat both aforementioned sub-experiments from another machine in the CrySP RIPPLE Facility at the University of Waterloo, which we refer to as *remote machine 1*.[1] In other words, the HTTP requests are generated on the browser extension as for the previous sub-experiments but are sent from this remote machine through the curl tool. We measure and plot the same six time measurements as for the previous sub-experiments wherein the requests were sent from the localhost interface of the servers' machine.

[1]The round-trip time reported by the `curl` tool for an HTTP request from this machine to the control server machine was found to be about 6 ms.

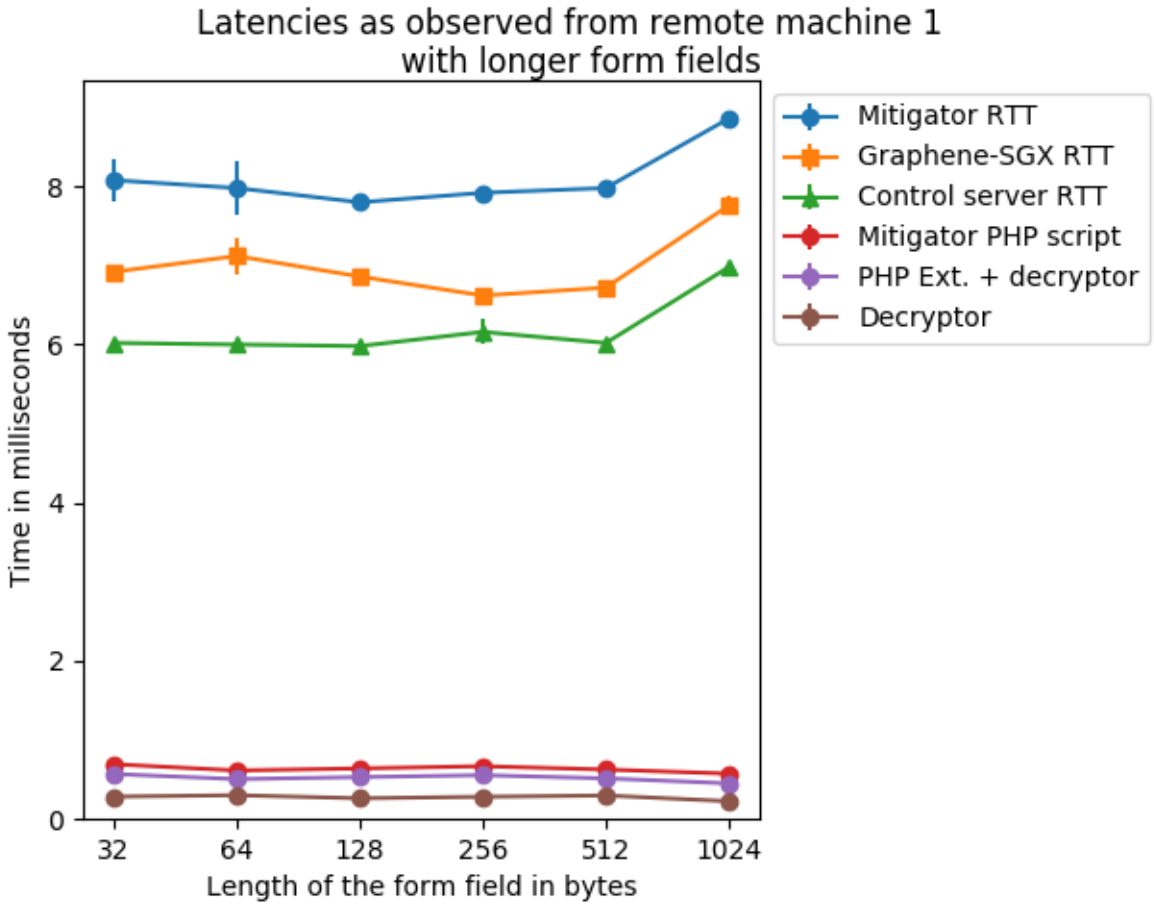

Figure 6.3: Measurement of server-side latency from remote machine 1 as the size of a form field increases: the length of a single form field was doubled in steps from 32 to 1024 and we measured the round-trip times (RTT) from remote machine 1 for each of the Mitigator, Graphene-SGX, and control servers, and the averages, over 50 requests for each form field length, are plotted in blue, orange, and green respectively. For the Mitigator server, we measured the total wall clock time spent within the PHP script, the total time spent within the PHP extension and the decryptor enclaves and the time spent within the decryptor enclave and the corresponding averages are plotted in red, light purple, and brown respectively. The standard error bars for most values in this dataset are too small to be seen on this graph.

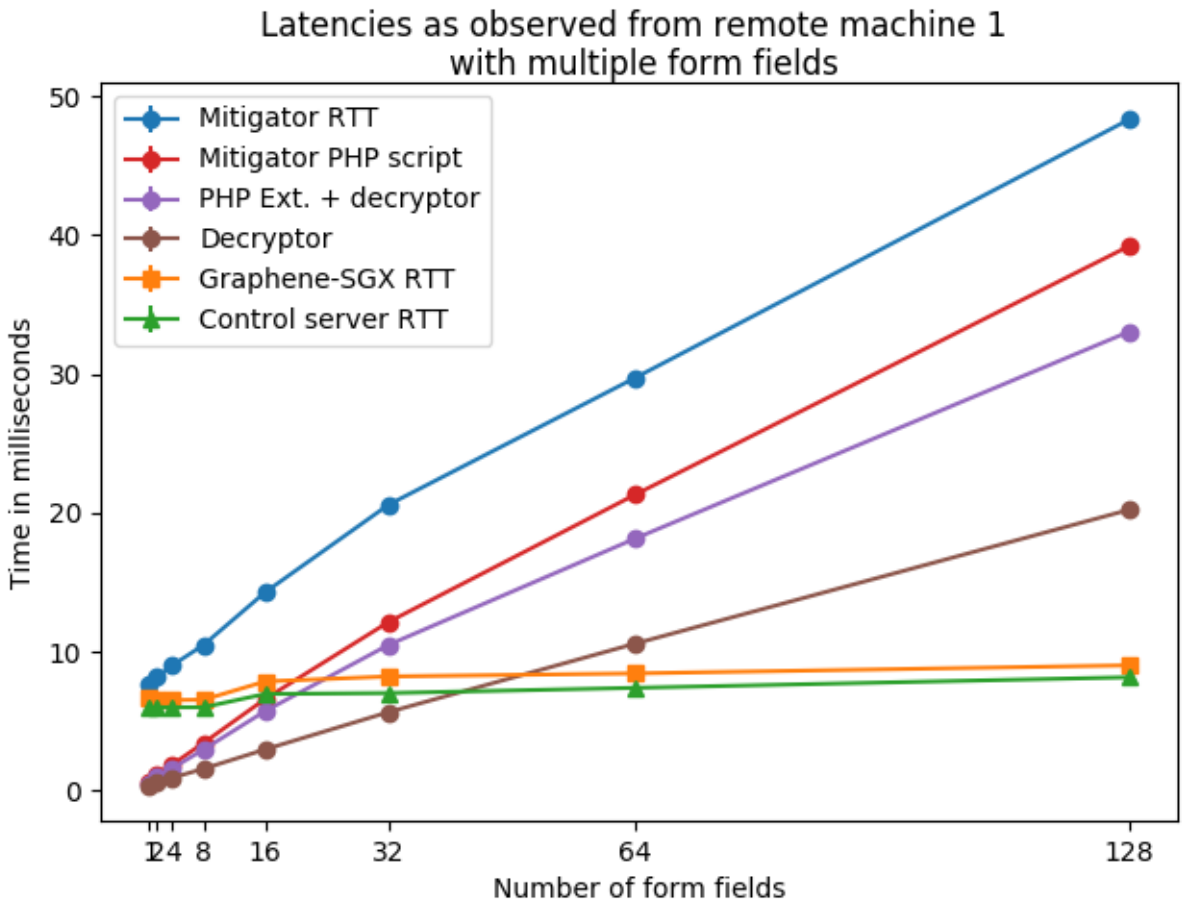

Figure 6.4: Measurement of server-side latencies from remote machine 1 as the number of form fields increases: starting from 1, the number of form fields was doubled in steps up to 128, and we measured the round-trip times (RTT) from remote machine 1 for each of the Mitigator, Graphene-SGX, and control servers, and the averages, over 50 requests for each number of form fields, are plotted in blue, orange, and green respectively. For the Mitigator server, we measured the total wall clock time spent within the PHP script, the total time spent within the PHP extension and the decryptor enclaves, and the time spent within the decryptor enclave and the corresponding averages are plotted in red, light purple, and brown respectively. The standard error bars for this dataset are too small to be seen on this graph.

From Figure 6.3, we can observe that any increases in the total computation time within enclaves for *longer* form fields are not significant with respect to even a small network overhead, as is the case for a client on our remote machine in the RIPPLE facility. (We again observe that as there are two TCP packets for HTTP requests with a form field of length 1024, the round-trip times for all three servers increase significantly.) However, from Figure 6.4, one can see that as the *number* of form fields increases, the corresponding increases in the total computation time within enclaves (shown in light purple) contribute to a significant fraction of the total Mitigator network round-trip time (shown in red). The total computation time within enclaves increases linearly in the number of form fields, as was the case in Figure 6.2. Again, we expect that the optimizations which we outline in Section 6.2 should render this computation time insignificant with respect to the total network round-trip time.

6.4 End-to-end latency

In the set of experiments in Section 6.3, we measured the average runtime latency for *server-side* Mitigator components; that is, for the decryptor and target enclaves. However, a typical user running Mitigator may face significant latencies due to Mitigator's client-side components. We now proceed to measure the end-to-end latency of Mitigator; this includes the latency on account of encryption of the form fields in the browser extension in addition to the network and server-side latencies. For the set of experiments in this subsection, we generate and send form field requests from a browser extension that runs on another remote machine, which we refer to as remote machine 2.[2] As discussed in Section 6.1, we modified the browser extension to send these requests to the Mitigator, Graphene-SGX, and control servers and as we described in Section 6.2, we instrumented it to measure the encryption time and the total round-trip time, which starts from when the extension sends a request to the second PHP page for the respective server and ends when it obtains a complete response.

We repeat subexperiments 1 and 2 as in Section 6.3 and from the browser extension on remote machine 2, we send $n = 10$ requests for each value of the independent variable; that is, the size of the form field sent in a request or the number of form fields sent per request. We arrange the timestamps recorded in the browser extension and in the server-side components for each request in a chronological order and subtract the client's encryption start timestamp from every other timestamp. We then sort each run of the experiment for a given value of the independent variable by the last timestamp for each run; that is, the time when remote machine 2 receives the complete response. We choose the median run for each value of the independent variable as a

[2]The round-trip time reported by the `curl` tool for an HTTP request from this machine to the control server machine was found to be about 10 ms.

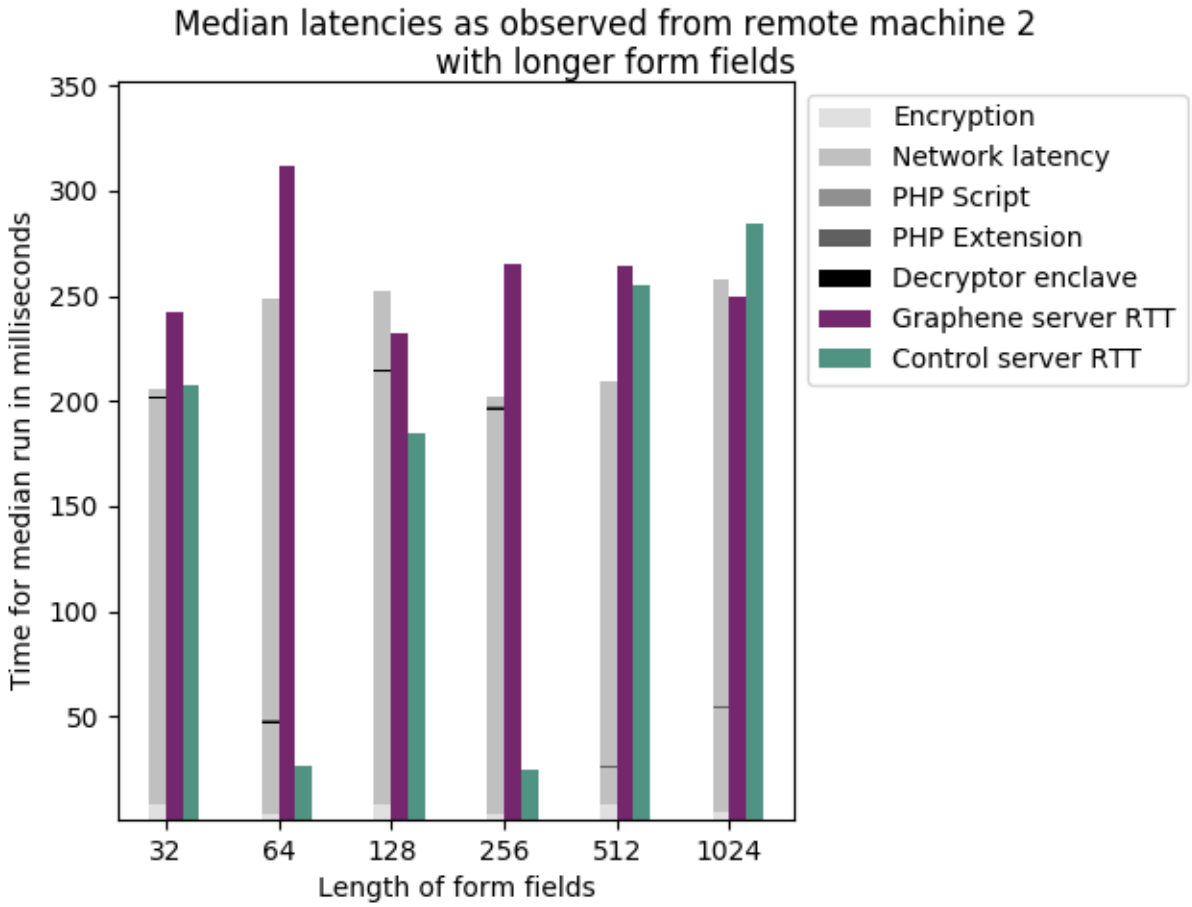

Figure 6.5: Distribution of time in median runs as the length of a single form field sent in the request varies: on doubling the length of the form field sent in each request, the client-side encryption time (lightest grey) and time spent in each of the Mitigator server-side components (darker shades of grey) are plotted for the median run for the given length of the form field. The median runs for the Graphene-SGX and control servers are also plotted in purple and teal respectively.

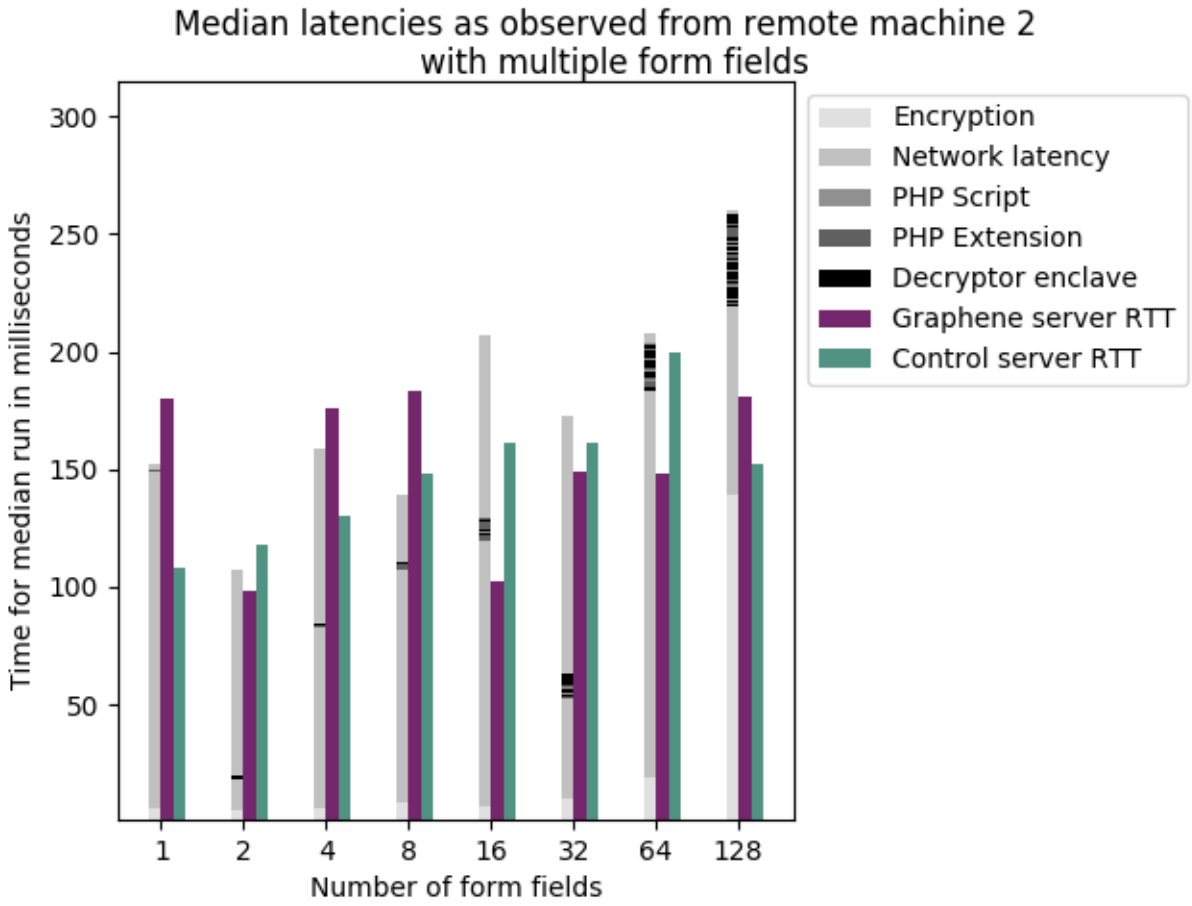

Figure 6.6: Distribution of time in median runs as the number of form fields sent in the request varies: on doubling the number of form fields sent in each request, the client-side encryption time (lightest grey) and time spent in each of the Mitigator server-side components (darker shades of grey) are plotted for the median run for the given number of form fields. The median runs for the Graphene-SGX and control servers are also plotted in purple and teal respectively.

representation of the latency that a user may expect to experience. In Figure 6.5 we plot the time spent in the browser extension and in each server-side component of Mitigator in a median run for each value of the form field size. We remark that due to the instrumentation of the browser extension, the network latency includes the time spent within the browser extension to prepare the request and to send it but does not include the latency for rendering an HTML response page. In Figure 6.6 we plot the same value for each value of the number of form fields sent in a request.

It is evident from Figure 6.5 that the client-side encryption time is very small in comparison to the total network overhead for Mitigator as the length of a form field increases. From Figure 6.6, this overhead remains small as the number of form fields to be encrypted increases up to 64 form fields. This observation further substantiates our claim that Mitigator does not incur large overheads for a remote client. The total round-trip time for requests to the Mitigator server is approximately the same as that for requests to the Graphene-SGX server as the length of the form field increases. However, from Figure 6.6, this round-trip time increases significantly as the number of form fields increases.

6.5 Summary

We have conducted experiments to determine the runtime latency of Mitigator as a significant runtime latency could deter usability. We found that the total round-trip time for a Mitigator-supporting server is significant in comparison to a Graphene-SGX or plain Apache server only if the network latency is very low. Under reasonable network latencies, as is to be expected for a remote client, we found that the difference in the round-trip time for a Mitigator-supporting server versus a Graphene-SGX server is minimal. However, about twice as much latency overhead is incurred with respect to a control server which is not run within the Intel SGX platform. In particular, although the overhead with respect to the Graphene-SGX and control servers does not increase linearly with longer form fields, it does increase linearly with a higher number of form fields. We propose a simple optimization in the implementation to eliminate this linear overhead. We leave this optimization to future work. We have thus measured and determined the primary sources of latency in our implementation, as per our goal for this chapter.

Chapter 7

Future work

There are many aspects of our design and implementation that can can be improved upon in future work and there are some implementation aspects that can be extended or generalized.

We begin with a discussion of systemic improvements to the design and implementation of server-side enclaves in each of the three verification, post-verification, and runtime stages. First, as we have only implemented simple privacy policy statements in the static code analysis tool within the verifier enclave, further work needs to be done to understand what kinds of statements can be checked using a static analysis tool. Additionally, we have only used a hard-coded configuration file of the static code analysis tool as the privacy policy for our proof-of-concept implementation and in this sense, such a privacy policy is not reflective of the complexities in real-world textual privacy policies. Therefore, another avenue of future work is to explore how to use the output privacy policy model from existing and future automatic privacy policy natural language processing tools to configure the static code analysis tool in the verifier enclave. On the other hand, our work also opens up an interesting possibility of using the output of a static source code analysis tool to guide developers to forming a fine-grained privacy policy that reflects the source code. For the post-verification and runtime stages, hosting multiple decryptor and target enclaves, possibly behind a load balancer, may be beneficial to website providers as it would decrease the fraction of the total client traffic that each target and decryptor enclave processes. Therefore, extending our design and proof-of-concept implementation to a distributed setting, that is, to cater for multiple decryptor and target enclaves on different machines, would make Mitigator more robust and deployable in real-world organizations.

In terms of extensibility, our work benefits significantly from research into running applications within trusted hardware platforms. Shim libraries that provide support for features listed in Table 2.1 can be used to deploy server-side enclaves. In more general terms, our design is not

limited to Intel SGX and thus further research into implementing our design on trusted hardware platforms that provide attestation and sealing-like properties would mean that website providers can deploy a Mitigator-like system on top of any existing trusted hardware platforms that they possess.

Last but not least, as we mentioned earlier, it is very important for privacy-enhancing technologies to be developed and deployed such that users *want* to use them. Conducting a thorough usability study to determine whether different users *want* to use Mitigator's browser extension, and if so, if the extension is *usable* or user-friendly, is thus a relevant line of future work. Our system has been designed to provide users guarantees that the processing of their data on server-side code is in compliance with the privacy policy, without revealing the source code to users or third parties. However, modern websites use dynamic Javascript code to process users' data on their machines; assuring compliance of client-side code is not within our problem statement but remains a very relevant adjacent problem, especially for the usability of our tool. In other words, users may reasonably expect a tool like ours to also provide guarantees over client-side code.

Chapter 8

Conclusion

In this work, we sought to enforce compliance of a website's source code with its privacy policy and signal the presence of this compliance in a trustworthy manner to users. We aimed to show that:

It is possible to build a system that can provide a verifiable guarantee to a user that the personal data that they submit to a website can only be processed by code that has been verified to be compliant with the website's privacy policy. The guarantee is verifiable in that the user can verify the integrity of the code that does this compliance check.

In this thesis, we have provided a design and outlined a proof-of-concept implementation, Mitigator, for the system we claimed could be built in the thesis statement. We have discussed relevant design choices for the security of our system and have detailed our reasoning for the correctness of our implementation of each of our server-side and client-side components. Additionally, we have evaluated the performance of our system and identified relevant sources of latency. Finally, we have described avenues of future work that are relevant to furthering the scope and applicability of our work.

We hope that Mitigator opens up further opportunities for research and development of prototypes to ensure that source code that processes users' data is compliant with written guarantees that are provided to users.

References

[Adv18] Advanced Micro Devices. *Secure Encrypted Virtualization API Version 0.17*. Technical Preview. Advanced Micro Devices, 2018. URL: https://www.amd.com/system/files/TechDocs/55766_SEV-KM_API_Specification.pdf.

[AHK+03] Paul Ashley, Satoshi Hada, Günter Karjoth, Calvin Powers, and Matthias Schunter. *Enterprise Privacy Authorization Language (EPAL 1.2)*. English. 2003-11. URL: https://www.w3.org/Submission/2003/SUBM-EPAL-20031110/#epalinuse (visited on 2019-02-20).

[AKO+17] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. *TaLoS: Secure and Transparent TLS Termination inside SGX Enclaves*. Technical Report. Imperial College London, 2017.

[ATG+16] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. "SCONE: Secure Linux Containers with Intel SGX". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 689–703. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

[BGR+18] Eleanor Birrell, Anders Gjerdrum, Robbert van Renesse, Håvard Johansen, Dag Johansen, and Fred B. Schneider. "SGX Enforcement of Use-Based Privacy". In: *Proceedings of the 2018 Workshop on Privacy in the Electronic Society*. WPES'18. Toronto, Canada: ACM, 2018, pp. 155–167. ISBN: 978-1-4503-5989-4. DOI: 10.1145/3267323.3268954. URL: https://dl.acm.org/citation.cfm?id=3268954.

[BJG+18] Andrea Bischel, Justinha, Ed Gallagher, Liza Poggemeyer, and Duncan Mackenzie. *Trusted Platform Module Technology Overview*. 2018. URL: https://docs.microsoft.com/en-us/windows/security/information-protection/tpm/trusted-platform-module-overview (visited on 2019-04-19).

[BRS+17] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. "Efficient and Flexible Discovery of PHP Application Vulnerabilities". In: *2017 IEEE European Symposium on Security and Privacy (EuroS P)*. 2017, pp. 334–349. DOI: 10.1109/EuroSP.2017.14.

[BS14] Travis D. Breaux and Florian Schaub. "Scaling requirements extraction to the crowd: Experiments with privacy policies". In: *2014 IEEE 22nd International Requirements Engineering Conference (RE)*. 2014, pp. 163–172. DOI: 10.1109/RE.2014.6912258.

[BWK+17] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. "Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 1041–1056. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck.

[CD16] Victor Costan and Srinivas Devadas. "Intel SGX Explained". In: *Cryptology ePrint Archive* (2016). Report: 2016/086. URL: https://eprint.iacr.org/2016/086.

[CLD16] Victor Costan, Ilia Lebedev, and Srinivas Devadas. "Sanctum: Minimal Hardware Extensions for Strong Software Isolation". In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 857–874. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan.

[CLM+02] Lorrie Cranor, Marc Langheinrich, Massimo Marchiori, Martin Presler-Marshall, and Joseph Reagle. *The Platform for Privacy Preferences 1.0 (P3P1.0) Specification*. English. 2002. URL: https://www.w3.org/TR/P3P/ (visited on 2019-02-20).

[Cra12] Lorrie Faith Cranor. "Necessary but not sufficient: Standardized mechanisms for privacy notice and choice". In: *Journal on Telecommunications and High Technology Law* 10 (2012), pp. 273–308.

[CS13] Stephen Checkoway and Hovav Shacham. “Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface”. In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’13. Houston, Texas, USA: ACM, 2013, pp. 253–264. ISBN: 978-1-4503-1870-9. DOI: 10.1145/2451116.2451145. URL: http://doi.acm.org/10.1145/2451116.2451145.

[Dis14] Disconnect. *Privacy Icons*. 2014. URL: https://web.archive.org/web/20141024015454/disconnect.me/icons/ (visited on 2019-03-29).

[Dwo16] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication*. NIST Special Publication 800-38B. National Institute of Standards and Technology, 2016. DOI: doi:10.6028/nist.sp.800-38b. URL: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38b.pdf.

[EMV+16] Eslam Elnikety, Aastha Mehta, Anjo Vahldiek-Oberwagner, Deepak Garg, and Peter Druschel. “Thoth: Comprehensive Policy Compliance in Data Retrieval Systems”. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 637–654. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/elnikety.

[FHA+18] Michael Freyberger, Warren He, Devdatta Akhawe, Michelle L Mazurek, and Prateek Mittal. “Cracking ShadowCrypt: Exploring the Limitations of Secure I/O Systems in Internet Browsers”. In: *Proceedings on Privacy Enhancing Technologies* 2018.2 (2018), pp. 47–63. DOI: http://dx.doi.org/10.1515/popets-2018-0012.

[GLS+12] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. “Hails: Protecting Data Privacy in Untrusted Web Applications”. In: *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX, 2012, pp. 47–60. ISBN: 978-1-931971-96-6. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/giffin.

[Gol07] Ian Goldberg. “Privacy Enhancing Technologies for the Internet III: Ten Years Later”. In: *Digital Privacy: Theory, Technologies, and Practices*. Ed. by Alessandro Acquisti, Stefanos Gritzalis, Costos Lambrinoudakis, and Sabrina di Vimercati. 2007. URL: https://cs.uwaterloo.ca/~iang/pubs/pet3.pdf.

[Goo19] Google. *Protocol Buffers*. 2019. URL: https://developers.google.com/protocol-buffers/ (visited on 2018-03-09).

[GS05] Rachel Greenstadt and Michael D. Smith. "Protecting Personal Information: Obstacles and Directions". In: *4th Annual Workshop on the Economics of Information Security, WEIS*. 2005. URL: http://infosecon.net/workshop/pdf/48.pdf.

[HAJ+14] Warren He, Devdatta Akhawe, Sumeet Jain, Elaine Shi, and Dawn Song. "ShadowCrypt: Encrypted Web Applications for Everyone". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: ACM, 2014, pp. 1028–1039. ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660326.

[HFL+18] Hamza Harkous, Kassem Fawaz, Rémi Lebret, Florian Schaub, Kang G. Shin, and Karl Aberer. "Polisis: Automated Analysis and Presentation of Privacy Policies Using Deep Learning". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 531–548. ISBN: 978-1-931971-46-1. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/harkous.

[Int18] Intel Corporation. *Intel® Software Guard Extensions (Intel® SGX) Data Center Attestation Primitives: ECDSA Quote Library API*. en. 2018. URL: https://download.01.org/intel-sgx/dcap-1.0/docs/SGX_ECDSA_QuoteGenReference_DCAP_API_Linux_1.0.pdf (visited on 2018-09-27).

[Int19a] Intel Corporation. *GitHub - intel/linux-sgx: Intel SGX for Linux**. 2019-04. URL: https://github.com/intel/linux-sgx (visited on 2019-04-07).

[Int19b] Intel Corporation. *Intel® Software Guard Extensions (Intel® SGX) Developer Guide*. 2019. URL: https://download.01.org/intel-sgx/linux-2.5/docs/Intel_SGX_Developer_Guide.pdf (visited on 2019-04-07).

[Isa18] Isan-Rivkin. *Graphene don't support sealing/unsealing functions #207*. 2018-05. URL: https://github.com/oscarlab/graphene/issues/207 (visited on 2019-04-02).

[JBR+12a] Michiel de Jong, Jan-Christoph Borchardt, Hugo Roy, Ian McGowan, Jimm Stout, Suzanne Azmayesh, and Christopher Talib. *Terms of Service; Didnt Read*. 2012. URL: https://tosdr.org/index.html (visited on 2019-04-02).

[JBR+12b] Michiel de Jong, Jan-Christoph Borchardt, Hugo Roy, Ian McGowan, Jimm Stout, Suzanne Azmayesh, and Christopher Talib. *Topics - Terms of Service; Didnt Read*. 2012. URL: https://tosdr.org/topics.html (visited on 2019-04-02).

[JKK06] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. “Pixy: a static analysis tool for detecting Web application vulnerabilities”. In: *2006 IEEE Symposium on Security and Privacy (S P'06)*. 2006. DOI: 10.1109/SP.2006.29.

[KBCR09] Patrick Gage Kelley, Joanna Bresee, Lorrie Faith Cranor, and Robert W. Reeder. “A ”Nutrition Label” for Privacy”. In: *Proceedings of the 5th Symposium on Usable Privacy and Security*. SOUPS '09. Mountain View, California, USA: ACM, 2009, 4:1–4:12. ISBN: 978-1-60558-736-3. DOI: 10.1145/1572532.1572538.

[KK03] Larry Korba and Steve Kenny. “Towards Meeting the Privacy Challenge: Adapting DRM”. In: *Digital Rights Management*. Ed. by Joan Feigenbaum. Berlin, Heidelberg: Springer, 2003, pp. 118–136. ISBN: 978-3-540-44993-5. DOI: 10.1007/978-3-540-44993-5_8.

[KKP+18] Klaudia Krawiecka, Arseny Kurnikov, Andrew Paverd, Mohammad Mannan, and N. Asokan. “SafeKeeper: Protecting Web Passwords Using Trusted Execution Environments”. In: *Proceedings of the 2018 World Wide Web Conference*. WWW '18. Lyon, France: International World Wide Web Conferences Steering Committee, 2018, pp. 349–358. ISBN: 978-1-4503-5639-8. DOI: 10.1145/3178876.3186101.

[KMC11] Jayanthkumar Kannan, Petros Maniatis, and Byung-Gon Chun. “Secure Data Preservers For Web Services”. In: *Proceedings of the 2Nd USENIX Conference on Web Application Development*. WebApps'11. Portland, OR: USENIX Association, 2011, pp. 3–3. URL: http://dl.acm.org/citation.cfm?id=2002168.2002171.

[KSH12] Iacovos Kirlappos, M. Angela Sasse, and Nigel Harvey. “Why Trust Seals Don't Work: A Study of User Perceptions and Behaviour”. In: *Trust and Trustworthy Computing*. Vol. 7344. Berlin, Heidelberg: Springer, 2012, pp. 308–324. ISBN: 978-3-642-30920-5. DOI: 10.1007/978-3-642-30921-2_18.

[Kur18] Arseny Kurnikov. *SafeKeeper - Protecting Web Passwords using Trusted Execution Environments*. 2018. URL: https://github.com/SafeKeeper/safekeeper-chrome (visited on 2018-07-11).

[Kuv18] Dmitrii Kuvaiskii. *Fork Implementation in Graphene SGX*. 2018. URL: https://github.com/oscarlab/graphene/wiki/Fork-Implementation-in-Graphene-SGX (visited on 2019-04-02).

[Lew17] Truman Lewis. *TRUSTe pays penalty, stiffens standards in agreement with New York*. English. 2017. URL: https://www.consumeraffairs.com/news/truste-pays-penalty-stiffens-standards-in-agreement-with-new-york-040717.html (visited on 2019-03-29).

[LKS+19] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Kriste Asanovic, Dawn Song, Ilia Lebedev, Srini Devdas, Sagar Karandikar, and Albert Ou. *Keystone - Open-source Secure Hardware Enclave*. English. 2019. URL: https://keystone-enclave.org/ (visited on 2019-01-18).

[LPM+17] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. "Glamdring: Automatic Application Partitioning for Intel SGX". In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 285–298. ISBN: 978-1-931971-38-6. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind.

[LSG+17] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 557–574. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho.

[MAF+11] Petros Maniatis, Devdatta Akhawe, Kevin R Fall, Elaine Shi, and Dawn Song. "Do You Know Where Your Data Are? Secure Data Capsules for Deployable Data Protection." In: *13th Workshop on Hot Topics in Operating Systems*. Vol. 7. USENIX Association, 2011, pp. 193–205. URL: https://www.usenix.org/legacy/event/hotos/tech/final_files/ManiatisAkhawe.pdf.

[MC08] Aleecia M McDonald and Lorrie Faith Cranor. "The cost of reading privacy policies". In: *I/S: A Journal of Law and Policy for the Information Society* 4 (2008), p. 543. DOI: http://hdl.handle.net/1811/72839.

[McW02] Brian McWilliams. *Data Firm Exposes Records Online*. English. 2002. URL: https://www.wired.com/2002/01/data-firm-exposes-records-online/ (visited on 2018-05-13).

[Mer18] Sam Meredith. English. 2018. URL: https://www.cnbc.com/2018/04/10/facebook-cambridge-analytica-a-timeline-of-the-data-hijacking-scandal.html (visited on 2019-04-08).

[OTK+18] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 227–240. ISBN: 978-1-931971-44-7. URL: https://www.usenix.org/conference/atc18/presentation/oleksenko.

[Pod17] Guy Podjarny. *Which of the OWASP Top 10 Caused the World's Biggest Data Breaches?* English. 2017. URL: https://snyk.io/blog/owasp-top-10-breaches/ (visited on 2019-04-09).

[Ras10] Aza Raskin. *Privacy Icons: Alpha Release*. 2010. URL: http://www.azarask.in/blog/post/privacy-icons/ (visited on 2019-04-10).

[RBO+14] Edith Ramirez, Julie Bill, Maureen K. Ohlhausen, Joshua D. Wright, and Terrell McSweeny. *Data Brokers A Call for Transparency and Accountability*. Technical Report. Federal Trade Commission, 2014. URL: https://www.ftc.gov/system/files/documents/reports/data-brokers-call-transparency-accountability-report-federal-trade-commission-may-2014/140527databrokerreport.pdf.

[Sco19a] Scontain. *SCONE Background*. 2019. URL: https://sconedocs.github.io/background/#scone-approach (visited on 2019-04-07).

[Sco19b] Scontain. *SCONE Session*. 2019. URL: https://sconedocs.github.io/scone_session/#caveats (visited on 2019-04-07).

[Sco19c] Scontain. *Technical summary of Scone*. 2019. URL: https://sconedocs.github.io/technical_summary/#transparent-attestation (visited on 2019-04-07).

[Sco19d] Scontain. *Trusted DApps*. 2019. URL: https://sconedocs.github.io/dapps/#attesting-cas (visited on 2019-04-07).

[SGD+14] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K. Rajamani, Janice Tsai, and Jeannette M. Wing. "Bootstrapping Privacy Compliance in Big Data Systems". In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. SP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 327–342. ISBN: 978-1-4799-4686-0. DOI: 10.1109/SP.2014.28.

[SGF18] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. “ZeroTrace : Oblivious Memory Primitives from Intel SGX”. In: *Proceedings of the 25th Annual Network and Distributed Systems Security Symposium*. NDSS’18. San Diego, California, USA, 2018. URL: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018%5C_02B-4%5C_Sasy%5C_paper.pdf.

[SLTS17] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. “Panoply: Low-TCB Linux Applications With SGX Enclaves.” In: *Proceedings of the 24th Annual Network and Distributed Systems Security Symposium*. NDSS’17. San Diego, CA, USA, 2017. DOI: http://dx.doi.org/10.14722/ndss.2017.23500. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/panoply-low-tcb-linux-applications-sgx-enclaves/.

[SSL+17] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A. Seshia. “A Formal Foundation for Secure Remote Execution of Enclaves”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. Dallas, Texas, USA: ACM, 2017, pp. 2435–2450. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134098.

[SWH+16] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D Breaux, and Jianwei Niu. “Toward a framework for detecting privacy policy violations in android application code”. In: *Proceedings of the 38th International Conference on Software Engineering*. ACM. 2016, pp. 25–36. DOI: 10.1145/2884781.2884855.

[TPV17] Chia-che Tsai, Donald E. Porter, and Mona Vij. “Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX”. In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 645–658. ISBN: 978-1-931971-38-6. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

[Tru19a] TrustArc Inc. *TrustArc Website Monitoring Manager*. 2019. URL: https://www.trustarc.com/products/website-monitoring-manager/ (visited on 2019-04-10).

[Tru19b] TrustArc Inc. *TRUSTe Enterprise Privacy Certification*. 2019. URL: https://www.trustarc.com/products/enterprise-privacy-certification/ (visited on 2019-02-20).

[Tsa17] Chia-che Tsai. *attestation #46*. 2017-02-20. URL: https://github.com/oscarlab/graphene/issues/46#issuecomment-289553325 (visited on 2019-04-02).

[Tsa18a] Chia-che Tsai. *Is it possible to keep the unmodified Applications outside SGX while the libraries inside SGX? #61*. 2018-05-23. URL: https://github.com/oscarlab/graphene/issues/61#issuecomment-306586041 (visited on 2019-04-02).

[Tsa18b] Chia-che Tsai. *No physical memory support, process creation may be slow #232*. 2018. URL: https://github.com/oscarlab/graphene/issues/232#issuecomment-410408466 (visited on 2019-03-26).

[VMW+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, 2018.

[Whi19] Zack Whittaker. *Equifax breach was 'entirely preventable' had it used basic security measures, says House report*. English. 2019. URL: https://techcrunch.com/2018/12/10/equifax-breach-preventable-house-oversight-report/ (visited on 2019-04-09).

[WSD+16] Shomir Wilson, Florian Schaub, Aswarth Abhilash Dara, Frederick Liu, Sushain Cherivirala, Pedro Giovanni Leon, Mads Schaarup Andersen, Sebastian Zimmeck, Kanthashree Mysore Sathyendra, N. Cameron Russell, Thomas B. Norton, Eduard Hovy, Joel Reidenberg, and Norman Sadeh. "The creation and analysis of a website privacy policy corpus". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*. ACL 2016. Berlin, Germany: ACL, 2016. DOI: dx.doi.org/10.18653/v1/P16-1126. URL: https://www.aclweb.org/anthology/P16-1126.

[ZB14] Sebastian Zimmeck and Steven M. Bellovin. "Privee: An Architecture for Automatically Analyzing Web Privacy Policies". In: *23rd USENIX Security Symposium (USENIX Security 2014)*. San Diego, CA: USENIX Association, 2014, pp. 1–16. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/zimmeck.

[ZWZ+17] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, Roger Iyengar, Bin Liu, Florian Schaub, Shormir Wilson, Norman Sadeh, Steven M. Bellovin, and Joel Reidenberg. “Automated Analysis of Privacy Requirements for Mobile Apps”. In: *24th Network & Distributed System Security Symposium (NDSS 2017)*. NDSS 2017. San Diego, CA: Internet Society, 2017. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/automated-analysis-privacy-requirements-mobile-apps/.

APPENDICES

Appendix A

Attestation in Intel SGX

We briefly describe local and remote attestation within Intel SGX.

A.1 Local attestation

We begin with a discussion of a one-sided local attestation for enclave A to enclave B, based on Costan and Devdas' report [CD16] and the Intel SGX SDK codebase [Int19a]. We refer to enclave A as the *attesting* enclave and enclave B as the *target* enclave. For local attestation, the attesting enclave A can generate a *report* that attests to its enclave and signer measurements to the target enclave B, such that enclave B can only verify the report if it is on the same machine. Figure A.1 shows the series of SGX instructions that enclave A executes to generate the report and enclave B executes to verify it.

A local attestation report consists of a keyed block-cipher-based message authentication code (CMAC [Dwo16]) over the attesting enclave's enclave and signer measurements, its other SECS attributes, and a user-specified string D, which is known as the *report data* within the SGX SDK documentation. To ensure that only a target enclave on the same machine can verify the CMAC over the report, the key to the CMAC, known as the *report key*, is derived from hardware secrets as well as attributes in the SECS of the target enclave. In particular, the expected enclave measurement, ISVSVN, and ISVPRODID for the target enclave B are to be included in generation of the report key. Collectively, these SECS fields are referred to as the *target information*. The target enclave executes the SGX microcode `EREPORT` instruction with null values in its input registers to retrieve its own target information, which it can later send to any attesting enclave.

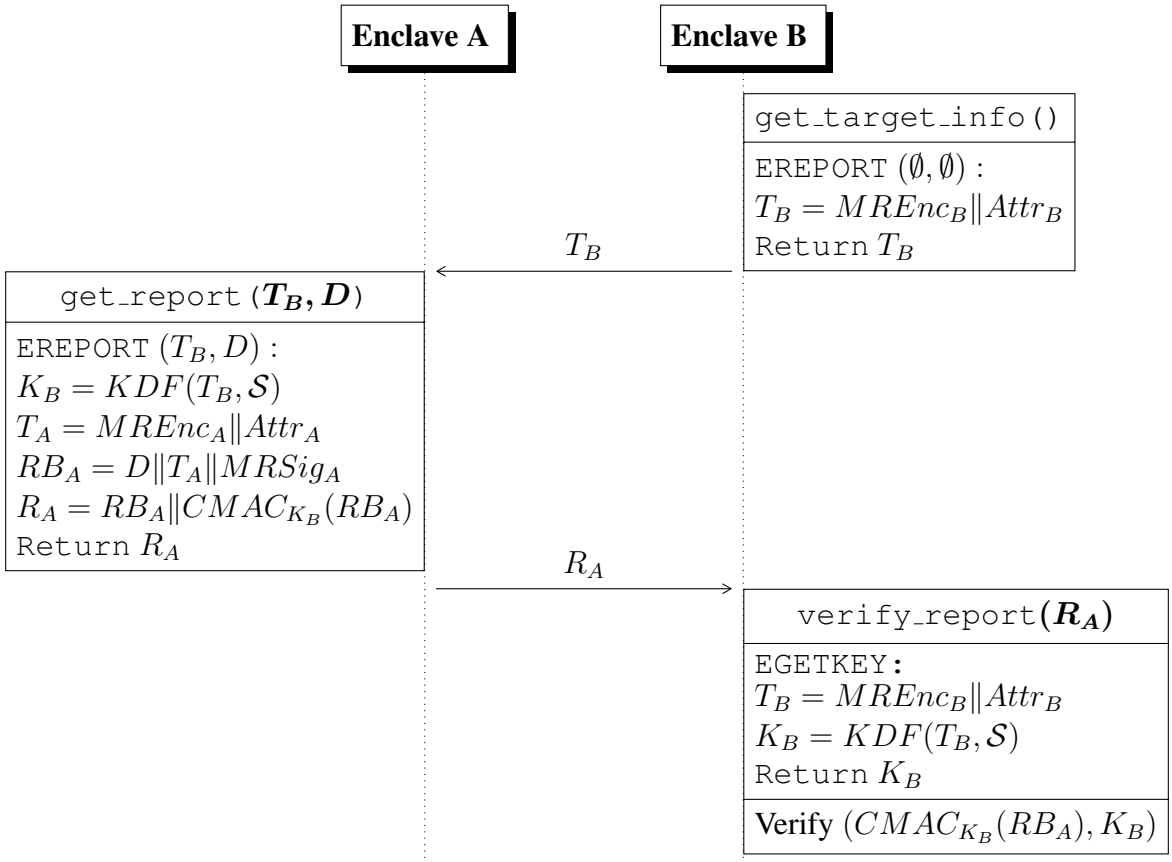

Figure A.1: Enclave A attests, with a data item D as the report data, to enclave B. At the end of this asymmetrical exchange, enclave B obtains an SGX report that attests to the enclave, signer measurements, and other attributes of enclave A, and contains the data item D. Both enclaves execute the `EREPORT` SGX instruction, whereas only the target enclave executes the `EGETKEY` SGX instruction. We remark that the `EGETKEY` SGX instruction invocation simply returns the report key. The report key is then passed to the enclave and used within an SGX SDK function to recompute the expected CMAC on the report and compare it to the one sent alongside the report. **Notation:** All operations performed within the enclaves, including the execution of these instructions, are encapsulated into one of three functions: `get_target_info()`, `get_report()`, and `verify_report()`. These functions are used as building blocks later on. $Attr_A$ refers to the SECS attributes of enclave A that are included in the generation of the report key, other than the enclave measurement. It therefore includes the ISVSVN and the ISVPRODID. $\mathcal{S}$ refers to machine-specific secrets that are burned into the CPU at manufacturing time and are passed as inputs to the derivation of the report key.

We denote this particular invokation of the EREPORT instruction with null arguments as the get_report() call.

The attesting enclave A then uses the target information T_B that was sent to it as the first message to generate a report targeted towards the target enclave B, as follows. It executes the EREPORT instruction, by passing the target information (T_B) and the report data D to its input registers. Internally, when executed with the above inputs, this instruction uses the target information and internal hardware-infused secrets, denoted by $\mathcal{S}$, to derive the report key K_B targeted towards enclave B. It then concatenates the report data D with the following attributes from its SECS page: the enclave measurement ($MREnc_A$), other attributes, such as the ISVSVN and ISVPRODID (encapsulated as $Attr_A$), and the signer measurement ($MRSig_A$). Finally, it computes a CMAC over the resulting data block, known as the *report body*, with the report key K_B to form the report R_A. We denote this particular invokation of the EREPORT instruction as the get_report() call.

The target enclave B then executes the EGETKEY instruction to retrieve the report key K_B. This instruction uses hardware secrets as part of the inputs to derive the key. Thus, this instruction will return the same key as the one used by the EREPORT instruction to generate the report on the attesting enclave, namely $\mathcal{K}_\mathcal{B}$, only if the target enclave is being run on the same machine as the attesting enclave (implying that $\mathcal{S}$ is the same). This instruction also uses the target information for the invoking enclave, that is, target enclave B, based on its SECS page attributes (T_B) as inputs to the derivation of the report key. The target enclave then uses the returned key K_B to recompute the CMAC over the data block RB_A in report R_A and to check that it is the same as the one in that report. We encapsulate the extraction of the report key and the verification of the report through the verify_report() function. Through the exchange of messages in Figure A.1, the target enclave obtains the correct enclave and signer measurements of the attesting enclave in the report body RB_A, and report data D for future usage. The target enclave B may authenticate the attesting enclave A based on these measurements. It may expect data item D to be of a certain form and therefore, proceed to verify it before using it. We refer the reader to §5.8.1 of Costan and Devdas' report for details on the EREPORT and EGETKEY instructions for local attestation.

We now proceed to discuss how a two-sided local attestation handshake can be used to establish a shared secret and therefore, a secure channel between two enclaves on the same machine. We base this discussion on the design in Intel SGX SDK's codebase for local attestation [Int19a]. We start with an initiator enclave B and a responder enclave A, which generate the short-term keypairs (b, g^b) and (a, g^a) respectively at the start of this protocol. The initiator enclave first executes get_target_info() to obtain its target information, say T_B. It can execute this function before generating its keypair and can reuse it for all responder enclaves. It then sends its public key g^b and the target information T_{ini} as the first message M_1.

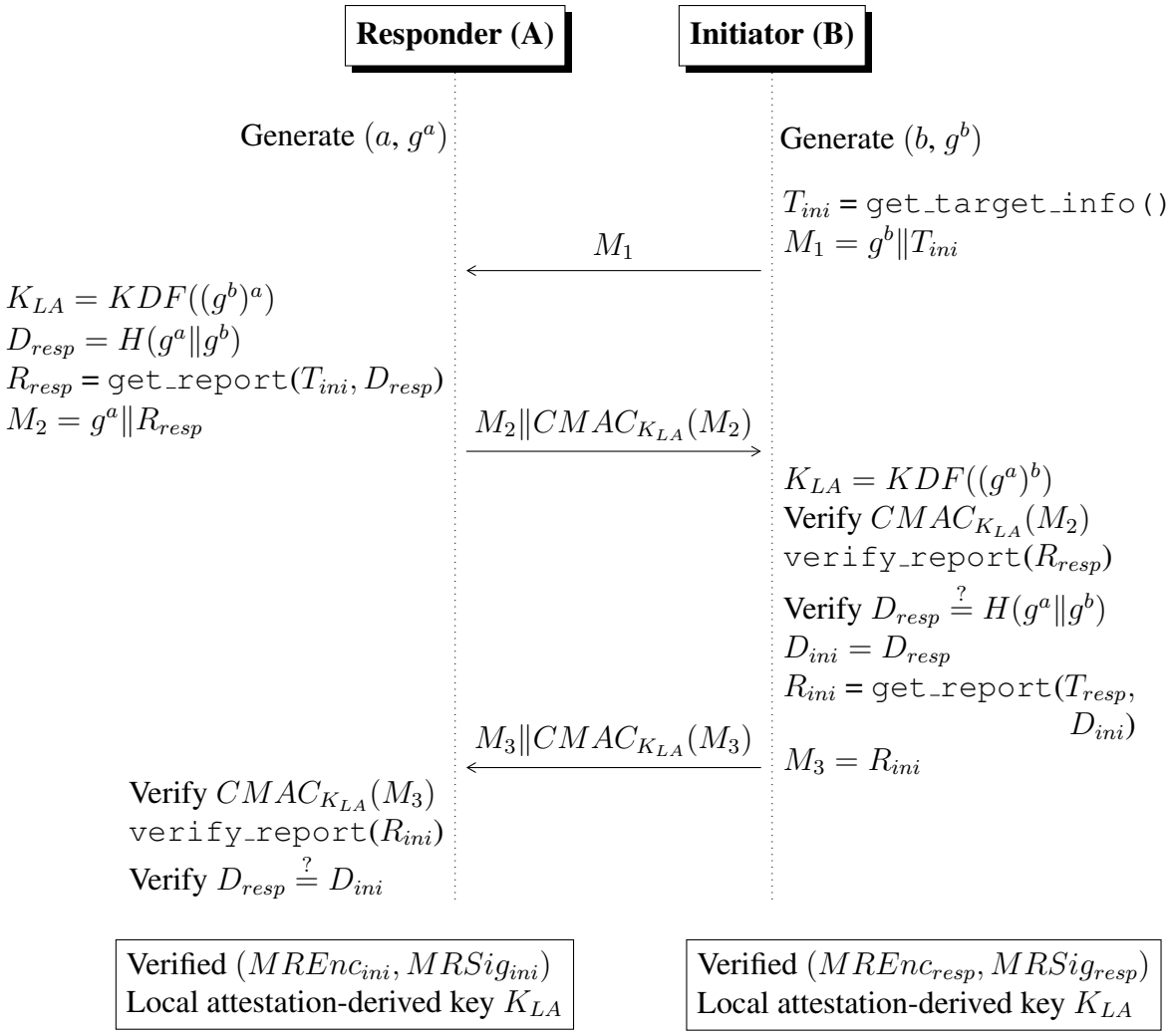

Figure A.2: Two-sided local attestation-based key exchange protocol used in Intel's SGX SDK and in Mitigator that shows how trusted Intel SGX instructions can be used to perform a bidirectional local attestation resulting in a secure channel between enclaves. The functions shown in `typewriter font` are executed through trusted SGX instructions, as shown in Figure A.1. As per the output of the `get_report()` function in that figure, the reports R_{ini} and R_{resp} always include information about their own SECS page in their report bodies, that is, $T_{ini} \| MRSig_{ini}$ and $T_{resp} \| MRSig_{resp}$ respectively.

Upon receiving the first message, the responder enclave uses the initiator enclave's public key g^b in that message and its own private key a to compute the shared secret g^{ab}. It consequently derives a symmetric key K_{LA} from the shared secret. It then computes a cryptographic hash over a concatenation of its own and the initiator's public key to generate the report data, D_{resp}, for its report to the initiator enclave. Including the initiator's public key g^b in the report data for a responder enclave's report ensures that the untrusted OS cannot simply replay back an older report, that was also targeted towards the initiator enclave, in the second message. In other words, it ensures freshness of the report in the second message. The responder enclave obtains a report for the initator enclave by calling the `get_report()` function with the target information of the target enclave, T_{ini}, which it obtained in the first message, as input arguments, and report data D_{resp}. Finally, the responder enclave generates a message M_2 that consists of its own public key and the report and computes a CMAC over it, using the key K_{LA}. It concatenates the CMAC to the message and sends it to the initiator enclave.

The initator enclave uses the responder enclave's public key g^a in the second message M_2 to generate the shared secret and derive the symmetric key, K_{LA}. It performs three integrity checks over the message before proceeding to generate its own report for the resonder enclave. First, it ensures that the first message was not altered by the untrusted OS, by verifying the CMAC over the message using key K_{LA}. In other words, if the target information of the responder enclave were to be modified from T_{resp} to some $T_{resp'}$ and/or if the responder enclave's public key were to be modified from g^a to $g^{a'}$, then the initiator enclave would find that the CMAC cannot be verified. Secondly, the initiator enclave verifies that the report R_{resp} was generated by another enclave on the same machine by calling the `verify_report` function over it. This check would succeed only if the initator enclave was on the same machine as the responder enclave (implying that the hardware secrets used in the derivation of the initiator's report key are the same) and if the responder enclave obtained and used the correct target information T_{ini}. Finally, following a successful result of the above function, the initiator enclave ensures that the report data is a hash of the concatenation of both enclaves' public keys: this ensures that the responder enclave indeed included both keys in the computation of the report data and detects attacks wherein the untrusted OS changes both enclaves' public keys (g^a and g^b) such that the same derived key K_{LA} is obtained, as the hash will differ if any of the keys differ. This concludes the verification of the second message.

The initiator enclave generates and sends a report in the last message of this protocol. The report data D_{ini} for this report is the same as that included in the report sent by the initiator enclave and so $D_{ini} = D_{resp}$. The target information for the responder enclave, namely T_{resp}, is within the report body of the responder enclave's report, namely R_{resp}. The initiator enclave therefore generates the report R_{ini} by calling the `get_report` function with D_{ini} and T_{resp} as

inputs. It consequently sends the report R_{ini} and a CMAC over it, using the derived key K_{LA}, as the third message.

Upon receiving the third message, the responder enclave performs the same three verifications over the message M_3 as the initiator enclave did upon receiving the second message M_2. In other words, it ensures the integrity of the third message and that it has obtained a fresh report from a responder enclave on the same machine. At the conclusion of the handshake, both enclaves have a verified report that attests to the other enclave's enclave and signer measurements. They have also established a symmetric key K_{LA} for further secure communication.

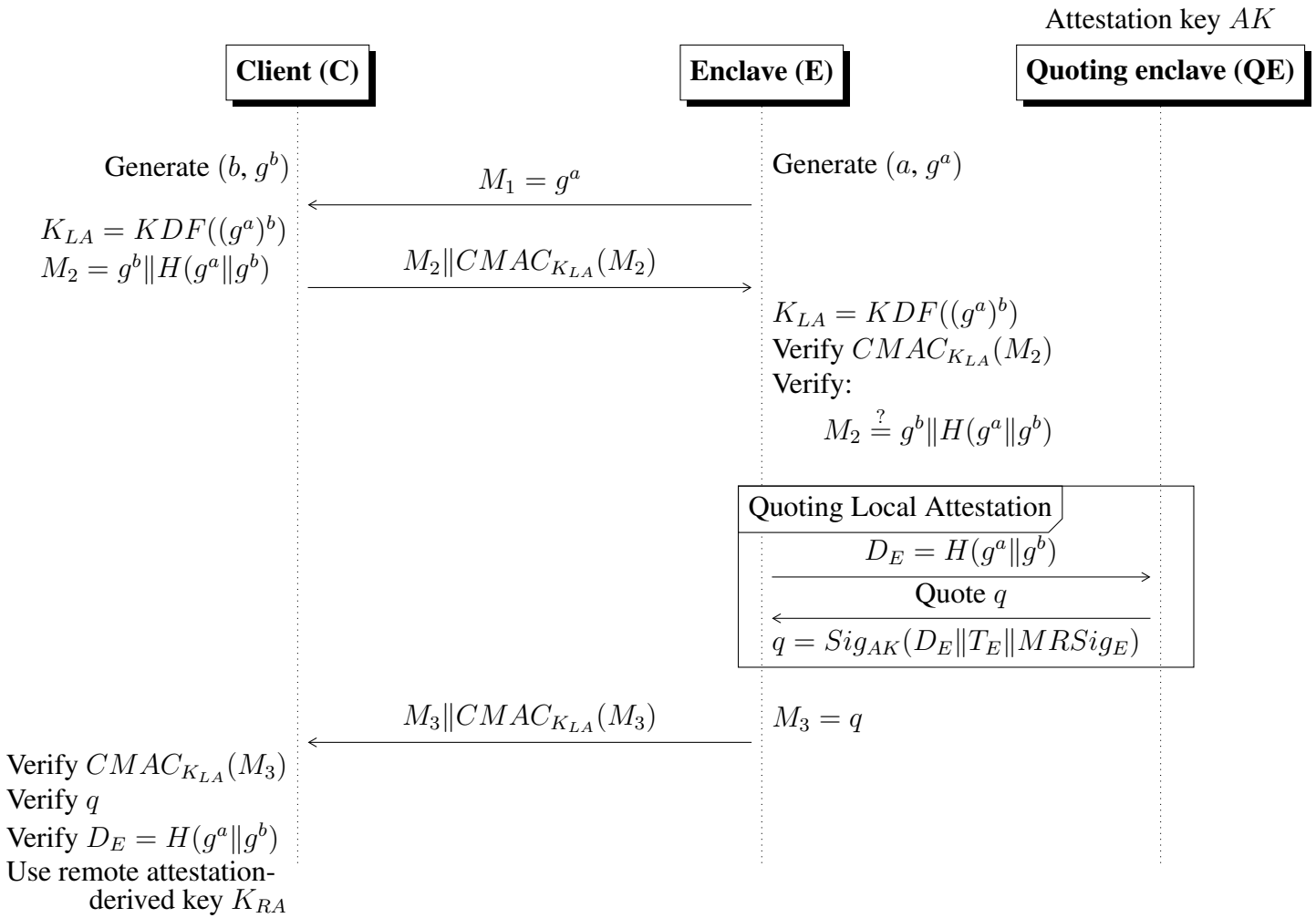

Figure A.3: An overview of the remote attestation protocol with a focus on the messages exchanged between the client and the enclave. The client verifies q by sending it to Intel to verify the signature and thereby obtains a verified signature over the enclave E's enclave and signer measurements ($MREnc_E, MRSig_E$). The quoting local attestation exchange between the enclave and the quoting enclave is shown in Figure A.4.

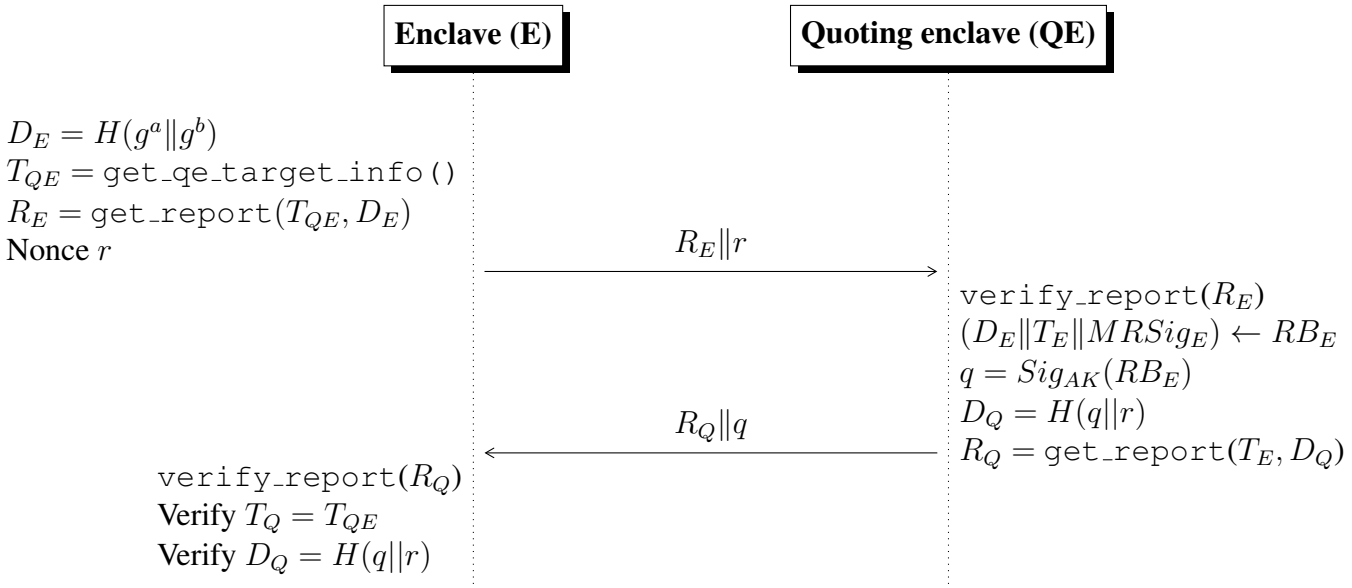

Figure A.4: This sequence diagram shows a custom local attestation protocol between the enclave E and another enclave with a special interface, known as the Quoting Enclave (QE), which occurs during remote attestation. The Quoting Enclave signs the report body of a local attestation report with an attestation key AK, such that the client can query Intel to verify the signature on the report body. We remark that the report body RB_E in the report R_E contains the report data D_E as well as the enclave and signer measurements and other enclave attributes ($T\_E \| MRSig_E$). The signed report body is known as a *quote* and is denoted by q in this diagram. We use the three functions shown in Figure A.1 and introduce another function named `get_qe_target_info()`, which returns the target information of the quoting enclave to the calling enclave.

A.2 Remote attestation

In the case of remote attestation, the client trusts the trusted hardware platform provider and wishes to ensure that their code is being run in an enclave on a trusted hardware platform. The trusted hardware platform provider acts as a root of trust to verify an enclave's report, known as a *quote*, that is passed to the client from the trusted hardware that hosts the remote enclave. A quote is simply a signature over a report generated on the given machine, with an *attestation key*. A special enclave known as the *quoting enclave* is used to provide quotes over reports. The attestation key is known to a remote attestation service hosted by Intel SGX, known as a *provisioning service*. Further information on the key derivation process for the report key and the verification of the report may also be obtained in §5.8.2 of Costan and Devdas' [CD16] report. We base the discussion below on the sample remote attestation code and libraries in Intel's SGX SDK codebase [Int19a]. We show the remote attestation in two diagrams for clarity and ease of reading: in Figure A.3 we focus on the interaction between the client and the enclave, whereas in Figure A.4 we focus on the interaction between the enclave and the quoting enclave.

The sample remote attestation protocol shown in Figure A.3 involves a simple Diffie-Hellman key exchange, as was the case in Figure A.2. The enclave and the client possess the keypairs (a, g^a) and (b, g^b) respectively. In the first message M_1, the enclave sends its public key to the client. The client computes the shared secret and derives the symmetric key K_{LA}. It then forms the second message M_2 by concatenating its own public key with a hash of both public keys, to ensure freshness of its generated message M_2 with updates in the enclave's public key g^a. The client then computes a CMAC over M_2 with the symmetric key K_{LA}. It sends M_2 concatenated with the above CMAC to the enclave over some (possibly insecure[1]) channel. Upon receiving the second message, the enclave first generates the shared secret and derives the symmetric key K_{LA}. It proceeds to verify the CMAC in the message using this key. Finally, the enclave ensures that the second message is not a replayed one by confirming that the rest of M_2 is a hash of both parties' public keys.

We note that until this point, no Intel SGX specific instructions have been executed and thus the client could inadvertently have established a channel with the untrusted OS. In the series of steps shown in Figure A.4, the enclave conducts local attestation with the Quoting Enclave (QE) in order to obtain a quote q that it then includes in the third message. We note that any enclave can obtain the target information of the quoting enclave, in order to generate a local attestation report targeted to it, which is encapsulated in Figure A.4 through a `get_qe_target_info()` function.

[1]The entire exchange between the client and the enclave could occur over TLS but as long as the TLS session key or certificate private key is known to the adversarial OS, this channel is an insecure channel for our case.

The enclave sets the report data D_E for the report to the quoting enclave to the hash of the concatenation of both parties' public keys and obtains the target information of the target enclave T_{QE} by executing the `get_qe_target_info()` function. The enclave passes T_{QE} and D_E as inputs to the `get_report` function and obtains the report R_E. It also generates a random nonce r to ensure freshness of the report generated by the quoting enclave. It sends the report R_E concatenated with the nonce r to the quoting enclave.

The quoting enclave verifies the report R_E through the `verify_report()` function call, as any other target enclave for a local attestation report would. It then generates a quote q by signing over the body of the verified report RB_E with the attestation key AK. It proceeds to generate the report data for its report to the given enclave, namely D_Q, by concatenating the quote q and the random nonce r, thereby ensuring that the report data changes in each response to a local attestation request from the same enclave (r) and that the report differs for different requesting enclaves and different clients (q). It generates a report by calling the `get_report()` function with the target information of the given enclave T_E, which is present in the verified report R_E, and the report data D_Q. It returns the quote q and the generated report R_Q to the requesting enclave.

The enclave first verifies the report, by calling the `verify_report()` over it. Following a successful verification, it performs two checks on the report body. First, it ensures that the report has been generated by the genuine quoting enclave on the same machine, by ensuring that the target information in the report body, namely T_Q, is the same as that which it obtained earlier as the return value of a call to the `get_qe_target_info` function. Secondly, it ensures that the report data is a hash of the quote q that it received from the QE and the random nonce r that it sent to the QE. Therefore, if the untrusted OS modified either the random nonce or the quote, then the enclave would detect this attack as the above check would fail. This concludes the local attestation of our given enclave with the quoting enclave. The enclave then sends the quote q, along with a CMAC computed over it using the key K_{LA}, to the client in the final message of the handshake with the client.

Upon receiving the third message from the enclave, the client first verifies the CMAC in this message, using the derived key K_{LA}. Secondly, the client verifies the quote q through Intel's remote attestation service. In an older version of Intel SGX, the client simply sends the request to this service and obtains a response. In an updated version [Int18], the attestation key AK is not bound to hardware secrets but can be generated randomly within any enclave that can provide the interface for a quoting functionality. A hash of the attestation key is placed in the quoting enclave report, which is in turn signed by a "Provisioning Certification Enclave" (PCE). The PCE has access to hardware-fused secrets for remote attestation that Intel can vouch for. In this new version, the client obtains a certificate chain that contains the quote q and that terminates at

a self-signed certificate for the PCE. The client also obtains an identifier that allows it to verify the certificate chain from Intel and thereby, it verifies the quote.

The client then ensures that the report data in the quote is equal to the hash of both enclaves' public keys. Effectively, this binds the quote to the current session of the Diffie-Hellman key exchange, thereby assuring the client that it is communicating with a genuine Intel SGX enclave, with the enclave and signer measurements $MREnc_E, MRSig_E$, which have been verified by a genuine quoting enclave.

Appendix B

Sealing-relevant algorithms

In this brief appendix, we expand on Algorithm 1 to list the algorithms for sealing a long-term keypair to disk and for unsealing such a keypair from the disk. One of these two functions is executed whenever the verifier or the decryptor enclave attempts to obtain its long-term keypair.

Algorithm 13 Algorithm for sealing a keypair.

global variables ▷ These variables are within the enclave.
$verification\_key$
$signing\_key$
end global variables

function ECALL_GENERATE_AND_SEAL_KEYPAIR()
$(signing\_key, verification\_key) \leftarrow create\_keypair()$
$plaintext\_keypair \leftarrow (signing\_key, verification\_key)$
▽ This is an internal call to a function in the SGX SDK to seal any data.
$sealed\_keypair \leftarrow sgxsdk\_seal(plaintext\_keypair)$
return $sealed\_keypair$
end function

Algorithm 14 Algorithm for unsealing a keypair.

function ECALL_UNSEAL_AND_SET_KEYPAIR($sealed\_keypair$)
    $\triangledown$ This is an internal call to a function in the SGX SDK to unseal any data.
    $(unseal\_status, plaintext\_keypair) \leftarrow sgxsdk\_unseal(sealed\_keypair)$
    if $unseal\_status == False$ **then**
        return $False$
    end if
    $(signing\_key, verification\_key) \leftarrow plaintext\_keypair$
    return $True$
end function