Miti Mazmudar* and Ian Goldberg

Mitigator: Privacy policy compliance using trusted hardware

Abstract: Through recent years, much research has been conducted into processing privacy policies and presenting them in ways that are easy for users to understand. However, understanding privacy policies has little utility if the website's data processing code does not match the privacy policy. Although systems have been proposed to achieve compliance of internal software to access control policies, they assume a large trusted computing base and are not designed to provide a proof of compliance to an end user. We design Mitigator, a system to enforce compliance of a website's source code with a privacy policy model that addresses these two drawbacks of previous work. We use trusted hardware platforms to provide a guarantee to an end user that their data is only handled by code that is compliant with the privacy policy. Such an end user only needs to trust a small module in the hardware of the remote back-end machine and related libraries but not the entire OS. We also provide a proof-of-concept implementation of Mitigator and evaluate it for its latency. We conclude that it incurs only a small overhead with respect to an unmodified system that does not provide a guarantee of privacy policy compliance to the end user.

Keywords: privacy policies, trusted hardware platforms, Intel SGX

DOI Editor to enter DOI
Received ..; revised ..; accepted ...

1 Introduction

Internet users have been habituated to click on a checkbox that says something to the effect of "I have read and understood the privacy policy" when they register an account on a website. As these privacy policies are long and written in legalese, users are effectively expected to trust that a website follows reasonable data handling practices. Earlier works, such as the Platform for Privacy Preferences (P3P) standard [11], which was a machine-readable privacy policy format, attempted to automatically match websites' privacy policies against users' preferences. Browsers would ensure that a website that a user visited had a privacy policy that was *at least* as restrictive as the user's. However, as Cranor found [12, pp. 296–299], 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 and therefore, P3P policies lost their meaning.

More recently, Slavin et al. [27] and Zimmeck et al. [33] used a model generated by a privacy policy model generator [2, 7, 16, 31, 32] to check the output of source-code analysis tools for privacy policy violations. They identify smartphone apps and third-party libraries that send users' information across the network, in violation of their privacy policies. In light of this, it is important to ensure that *websites'* source code is compliant with respect to their privacy policies.

Indeed, several systems that are variations of the above basic design have been proposed to assess the compliance of back-end website source code with formal logic representations of privacy policies [25] or access control policies [13, 15]. However, such systems require an end user to trust that the operating systems on the website's back-end machines are free of any vulnerabilities. Recent data leaks [30] exemplify poor data handling practices of organizations and thus it is reasonable for a user to not trust the operating systems on a website's server-side machines.

Trusted hardware platforms enable establishing trust in a small part of the CPU on the remote host machine, while treating the rest of the hardware and the OS of the remote host as untrusted and compromised. Although several approaches [6, 20, 29] have used trusted hardware platforms to reduce the trusted computing base for ensuring compliance of server-side code with privacy policies, they do not provide the following important properties simultaneously:

1. Not require the users to trust the OS.
2. Protecting users' data end to end, from its point of generation in the browser to where it is being processed on the trusted hardware platform.

***Corresponding Author: Miti Mazmudar:** University of Waterloo, E-mail: miti.mazmudar@uwaterloo.ca
Ian Goldberg: University of Waterloo, E-mail: iang@uwaterloo.ca

3. Providing users with a verifiable guarantee of the above end-to-end protection.
4. Ensuring that noncompliant applications do not abort in a production environment, thereby avoiding poor user experiences with the service.
5. Supporting web applications that perform various operations on users' data such as sharing data, using it for a given purpose, and so on.

In this work, we leverage trusted hardware platforms to present a design, named *Mitigator*, that satisfies all of the above properties. In particular, Mitigator provides a cryptographic guarantee to a user not only that the website code has been checked by the aforementioned verifier program to be compliant with the privacy policy, but also that user's data will *only* be handled by code that has been checked by the verifier program.

Organizations have several incentives to deploy an implementation of Mitigator. Such an implementation can increase the transparency of their data handling practices to users, while minimizing the trust assumptions that users are required to make, from including the entire OS to just a small trusted hardware module and a set of libraries to support it. The implementation can thus be used as a differentiating feature of their online products and services, in order to draw in privacy-conscious users. In addition, it can also be used to illustrate that their code has certain verified properties that are required in applicable legislation.

In this paper, we start start with providing relevant background and related work in Sections 2 and 3, and outline our threat model in Section 4. We then present the following three contributions within the subsequent sections:

1. We present *Mitigator*, a cryptographic design that securely uses underlying trusted hardware platform primitives to provide the aforementioned properties (Section 5).
2. We briefly describe a proof-of-concept implementation of Mitigator on the Intel SGX trusted hardware platform (Section 6).
3. We evaluate our proof-of-concept implementation, in terms of overheads compared to an otherwise identical setup that runs on an untrusted operating system and does not provide Mitigator's privacy guarantees (Section 7).

2 Background

The Mitigator design combines existing work on privacy policy model generation and static source-code analysis to check the website back-end code for compliance with the privacy policy. Our novel contribution lies in building cryptographic protocols on top of trusted hardware platforms to provide a *guarantee* to end users that their data, from the time it leaves the browser, is *only* ever handled by code that passed this check. In this section, we first focus on trusted hardware platforms and describe which properties are needed for Mitigator. We then proceed to briefly describe source code analysis tools and modern policy model generation tools.

2.1 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 Guard Extensions (SGX) platform [17] and AMD's Secure Encrypted Virtualization (SEV)/Secure Memory Encryption (SME) [1].

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.

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 can deterministically compute a specific hash of the enclave executable, known as the *enclave measurement*. The developer then signs over this enclave measurement 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*.

On the untrusted host, enclaves are loaded into protected pages in the memory, known as the *Processor Re-*

served Memory (PRM). The untrusted OS cannot read these pages as it does not have the relevant hardware-bound keys. This gives us the aforementioned property of confidentiality of the program state. Further, the untrusted OS cannot load a different enclave into memory without changing its enclave measurement, thus resulting in integrity of the program. Finally, 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 will be using the enclave and signer measurements within our design.

In addition to the above guarantee over the correctness of enclave and signer measurements, the Intel SGX trusted hardware platform provides sealing and attestation guarantees, as follows, which we use in our system:

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*. Such a secret can be sealed such that only enclaves with an identical enclave measurement can unseal it, or only ones with an identical signer measurement, or only the sealing enclave itself.
2. **Attestation:** Attestation corresponds to an enclave on a given machine proving to either another enclave on the same machine (*local* attestation) or to a remote client (*remote* attestation), that it is indeed a particular SGX enclave. The proof for local attestation can only be verified by the participating enclaves if they are on the same machine, whereas in remote attestation, the client needs verification from Intel that the proof given by the enclave is valid. In addition, attestations result in a secure (AEAD) channel being established between the two enclaves (local) or the enclave and the client (remote). We remark that both styles of attestation are conducted in the presence of an untrusted OS that can observe and modify messages between enclaves or between an enclave and the client.

2.2 Source code analysis tools

Information flow analysis has been used by security researchers to analyse server-side scripts to detect vulnerabilities such as SQL injection and XSS attacks: if any user-specified inputs are passed to output functions without being sanitized first, then a vulnerability may exist. Jovanovic et al. [18] propose Pixy, an open-source static PHP taint analysis tool that uses the above approach for automatically identifying vulnerabilities due to a lack of sanitization in server-side PHP code. Subsequently, Backes et al. [5] proposed PHP-Joern to conduct scalable, automated analyses of PHP scripts to detect a wider range of security vulnerabilities, such as control flow-related vulnerabilities. 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 source code analysis tools, possibly for a different back-end scripting language, can easily be used in place of Pixy.

2.3 Modelling natural language privacy policies

With recent advances in natural language processing, textual privacy policies can be translated into accurate models, which have then been used to check source code for compliance. Andow et al. [2] and Harkous et al. [16] present PolicyLint and Polisis respectively, both of which automatically generate a model of a given privacy policy for a given website using NLP and machine learning techniques. Polisis and PolicyLint 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. For instance, Sen et al.'s [25] first-order logic representation of privacy policies, named Legalease, uses finite lattices to represent values—Polisis and PolicyLint can be extended to automatically generate Legalease representations for the verifier program.

3 Related work

Related work in this area has several different goals and therefore, we first present the following five criteria (reflected in Table 1) to meaningfully compare existing work with Mitigator.

1. **Low TCB:** Systems that do not require trusting the OS to be free of vulnerabilities are categorized as requiring a low trusted computing base (TCB). Even systems using trusted hardware platforms can be categorized as having a *high* TCB if their design allows the OS to observe users' data in plaintext.
2. **End-to-end:** Akin to Saltzer and Schroeder's concept of complete mediation, we expect systems to guarantee that users' data is handled as per their

policies, from the point where it is collected in the browser to the point where it enters an enclave or the data-processing program.

3. **User guarantee:** Systems that do protect users' data end to end should also provide the users' client with a verifiable guarantee that it is functioning correctly.
4. **Static:** Systems that use information flow analysis to determine whether the server-side program is compliant with some policy can be static or dynamic. Static analysis tools make this decision at *compile* time, and therefore the verified applications are never denied access to users' data at runtime. For this reason, we prefer static tools. Dynamic tools make the decision at *runtime* and can thus verify a larger subset of language features, but can result in applications failing or aborting at runtime.
5. **General operations support:** The system should support checking a range of operations that applications tend to perform on user data, such as sharing users' data, determining whether all operations on users' data meet a set of acceptable purposes, ensuring data is not retained for a longer time, etc., rather than just checking for a specific operation.

Furthermore, like these past approaches, we remark that our design has two other features that are useful for deployability. First, it does not have a single point of failure. In other words, Mitigator does not require a single authority to participate by being online during the compliance checks of all Mitigator-supporting websites. Second, as website providers may have proprietary code running on their back-end systems, we also do not require them to publish their code for another party to check its compliance. Indeed, in Mitigator, back-end code changes are never exposed to users, as long as the code remains compliant with the privacy policy. We begin with discussing related literature with static designs and then proceed to discuss works with dynamic designs.

Static systems. Sen et al. [25] present Legalease, a first-order representation of privacy policies that captures a range of operations, such as using data for a given purpose, data retention, sharing data, and so on. They also bootstrap an information flow analysis tool named Grok, which performs static analysis across an entire distributed system and outputs its results to a policy checker. The policy checker then compares this output against the Legalease encoding of the privacy policy to report programs that violate the latter. Interestingly, of all the works that we analyze in this section,

Table 1. Comparison of related work within policy compliance: ○ indicates no support, and ● indicates full support. Refer to the criteria in Section 3 for a description of column headers.

System	Low TCB	End-to-end	User guarantee	Static	General ops.
Grok [25]	○	○	○	●	●
Thoth [13]	○	●	○	○	●
SDC [23]	●	●	○	○	●
SGX Use Privacy [6]	○	○	○	○	●
SafeKeeper [20]	●	●	●	●	○
SDP [19]	●	●	●	○	●
Riverbed [29]	○	●	●	○	●
Mitigator (this work)	●	●	●	●	●

Grok is the only system that we know of that has a large-scale deployment (within Microsoft's Bing search engine back end [25]). This shows that *static* analysis tools have enough expressiveness for use in a real-life compliance checker. However, Sen et al.'s system only guarantees compliance of data that is already within an existing distributed data store, rather than protecting it from when it leaves the users' browser and it is thus not an end-to-end system. Second, they do not provide users with any verifiable guarantees.

Most closely related to Mitigator in the context of privacy-preserving browser extensions is Krawcieka et al.'s [20] SafeKeeper. The authors cater to the problem of protecting passwords against compromised servers. SafeKeeper, like Mitigator, provides evidence of its integrity to a remote user through the use of an authenticated trusted hardware enclave on the server side. Before they leave the browser, users' passwords are encrypted to this server-side enclave and thus their system is end-to-end. However, their core design involves intentionally modifying passwords in an irreversible manner and cannot be trivially extended to support checking *general* operations on users' data. 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. Finally, SafeKeeper does not perform any static or runtime checks on the website source code or the privacy policy and so it trivially satisfies our static criterion, whereas Mitigator explicitly checks compliance of varying code against a possibly varying privacy policy.

Dynamic designs. An alternative to static information flow analysis is checking code at runtime using a reference monitor. Reference monitors keep track of data items or types that an application has accessed in the past and use this information to determine whether future access to a given datatype will allow the application to make inferences that violate a given policy. The reference monitor will deny noncompliant applications access to users' data at runtime and as a result, in contrast with static designs such as ours, these designs run the risk of aborting users' logged-in sessions.

Elnikety et al.'s Thoth [13] enables implementing access control policies that are derived from user-specified settings across various unmodified server-side applications. Users' data is augmented with these policies when it is created and Thoth thus protects users' data end to end, starting at the browser. Thoth's reference monitor mediates all file-related system calls made by server-side applications and denies applications access to data, at runtime, if it violates the policy. However, Thoth is not designed to provide any guarantees to end users and it has a large TCB, as it requires the end user to trust the reference monitor, a kernel module, and the rest of the OS. Thoth is also not designed to provide any verifiable guarantees to an end user.

In their Secure Data Capsules framework, Maniatis et al. [23] envision running a reference monitor-based design similar to Elnikety et al.'s Thoth within a trusted hardware platform. As users are assumed to be running the applications locally, the SDC framework mediates the data from end to end. However, Maniatis et al. also do not provide a verifiable guarantee to end users. Furthermore, they do not implement their design, and thus fail to highlight subtleties in running unmodified applications within a trusted hardware platform.

Birrell et al. [6] implement a similar system that follows the use-privacy model, wherein acceptable purposes of use of a given data type are determined through a social process. Their reference monitor, which runs within an SGX enclave, mediates applications' access to a data store. Birrell et al.'s model assumes users' data is in this data store and is therefore exposed to the OS when it is being collected or sent to the data store. Their system thus does not mediate users' data in an end-to-end manner. Simply performing attestation between the reference monitor and the user's client does not address this leakage, as the attestation only guarantees that the reference monitor runs as expected, not that users' data has been protected from exposure (to the OS or otherwise). Furthermore, akin to Thoth and SDC, Birrell et al. do not propose any mechanisms to provide verifiable guarantees to the users. Therefore, Birrell et al.'s scheme cannot be extended trivially to mediate users' data in an end-to-end manner, let alone satisfying the verifiable guarantee criterion. Finally, their design allows the untrusted OS to observe plaintext data in transit, and therefore, despite using a trusted hardware platform, it does not meet our criterion of a low TCB.

Kannan et al.'s Secure Data Preservers [19] allow users to run their own implementations of various sub-service interfaces, known as preservers, on their own data and configurable policies, within trusted hardware platforms. A reference monitor also runs within the trusted hardware platform and ensures that the policies are followed. Users perform remote attestation with this reference monitor, and thereafter install their preserver on that machine. Therefore, users can obtain a verifiable guarantee that their preserver is running as expected. Although their system supports different preservers, such as aggregating or filtering preservers, as a preserver is created for *each* user, this design requires significant memory: using the authors' self-reported figures for the preserver size, as the PRM is very limited in size (90 MB), only a few preservers can be run on the same machine.

Wang et al.'s Riverbed [29] supports checking the compliance of web applications against user-specified policies, within a tamper-resistant TPM module. The TPM only provides integrity guarantees for the bootloader, the OS, and the software stack (including the Riverbed components) and does not provide confidentiality over the executed program, in contrast with trusted hardware modules such as Intel SGX. Hence, as Wang et al. assume that the OS is free of exploitable bugs and include its hash within the integrity measurement of the module, they follow a weaker threat model than ours. Although their instrumented interpreter *does* terminate noncompliant web applications at runtime (and thus does not satisfy our *static* feature), Riverbed reduces the number of users affected at runtime to only those whose policy is violated. When a user first loads a page, the Riverbed client conducts remote attestation with the server application and checks that the measurement matches the expected one. Thus Riverbed can assure its users that the server-side machines are running the Riverbed runtime. Riverbed then forwards the users' data along with its policy, and thus, Riverbed mediates users' data in an end-to-end manner.

If a static analyzer were to be included in Riverbed, then in order to provide an end-to-end guarantee to users, apart from authenticating the target binary through remote attestation, the client would need to authenticate that the analyzer indeed analyzed the attested current target binary. We thus expect that extending Riverbed to support static analysis tools with a low TCB would result in a design similar to ours.

4 Threat model

Mitigator only requires users to trust a small hardware module and the software that runs within it, but not the complete operating system. 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. Such an adversarial OS can also manipulate the filesystem and return values of system calls to whatever it desires. As the TLS connection may reasonably be terminated outside the enclave, the OS can modify any HTML content before displaying it to the client as well as replay any content encrypted by the client to a server-side enclave. (Although Aublin et al. [4] propose TaLoS to terminate TLS connections within the enclave, this requires the TLS private key be generated within the enclave and not exported elsewhere—a possible drawback for deployment.) Moreover, we restrict any attacks by an adversarial OS to exclude all side-channel based attacks [8, 22, 28]: much recent literature surrounding trusted hardware is directed at preventing these attacks [10, 21, 24]. We also assume that the adversary is bounded by the hardness of cryptographic problems.

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 noncompliant website as a Mitigator-approved website. The user should expect a positive signal of compliance from the Mitigator browser extension and importantly, should treat the lack of a positive signal of compliance as noncompliance. We mentioned previously that we assume that the website provider has incentives to deploy Mitigator, such as increasing its transparency to users. Therefore, it is not in their interests to perform any denial-of-service (DoS) attacks on their own deployment of Mitigator. We also assume that in case the attacks occur due to unforeseen circumstances, the organization will attempt to identify and stop them, so that their users can continue to see their Mitigator client's positive signal of compliance.

5 Design

We start with a program, called the *verifier*, that is run within an enclave, known as the verifier enclave. We refer to this execution of the verifier enclave as making up the *verification* stage of Mitigator. Given the privacy policy file and source code files for the target website as inputs, the verifier checks the source code files for compliance with the privacy policy text, as shown in Figure 1.

Following that, the verifier constructs an enclave called the *target enclave* containing the aforementioned files and the webserver executable. It uses a long-term signing-verification keypair $(\mathcal{SK}_V, \mathcal{VK}_V)$ to produce a signature the target enclave, *only if* it verifies compliance. The valid target enclave may now be executed using a trusted hardware platform.

Importantly, as the verifier *statically* analyses the target website's files, we do not need to run checks on user data passed to the *deployed* target program. Consequently, the verifier does not need to be running when the target enclave has been deployed. Whenever the target enclave source code or privacy policy changes, the verifier enclave needs to be re-run on the new target enclave. The new signed target enclave can then be deployed.

We assume that this verifier program is sound, but not complete. In other words, it will only produce a signature over the target if it is compliant with the privacy policy. The verifier program (but *not necessarily* the target website source code) is open-sourced, and its soundness can be checked by any party. We assume that as the organizations deploying an implementation of Mitigator have incentives to deploy it, the developers can structure or annotate their code such that the verifier does not falsely flag it as noncompliant, thereby navigating around the incompleteness of the verifier.

5.1 Strawman design

We motivate Mitigator's stages after verification by first explicating a strawman protocol. In the strawman protocol, before loading the webpage, the client performs two remote attestation handshakes with the verifier and target server-side enclaves to authenticate them. The

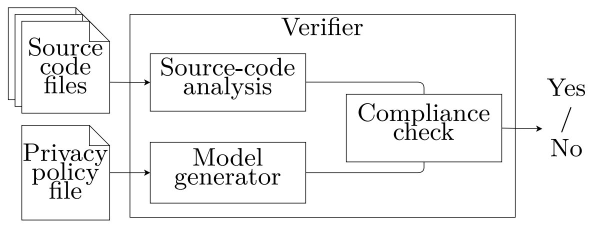

Fig. 1. An abstraction of the components used in existing literature to check compliance of source code with a privacy policy model. We discuss existing literature for source code analysis tools in Section 2.2 and generation of privacy policy models from their text in Section 2.3.

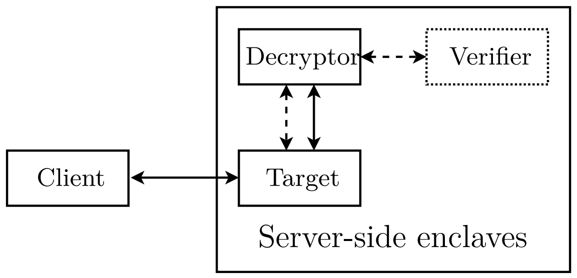

Fig. 2. A high-level block diagram of Mitigator. The component in the dotted box will be online in the verification stage. The dashed arrows indicate interactions occurring in the deployment stage, which are detailed in Figure 3. The solid arrows indicate interactions occurring in the runtime stage, which are detailed in Figure 4. The verifier essentially implements the algorithm shown in Figure 1, except that it runs within an enclave, and it signs the target enclave containing the source code files if they comply with the privacy policy.

client first performs remote attestation with the verifier enclave, ensuring that it has an enclave measurement equal to the one that the client expects from its open-sourced version. The verifier enclave sends the expected signer measurement of the target enclave through the remote attestation secure channel to the client. The client then performs a remote attestation handshake with the target enclave and authenticates it based on the signer measurement it obtained from the verifier enclave. The client finally sends its data over the remote attestation secure channel to the target enclave.

We outline two aspects of our strawman protocol that we improve upon in the Mitigator final design. First, we observe that the client learns whether the target enclave has been changed from the client's previous visit through its enclave measurement, which is included in the attestation report. As mentioned previously, website providers may be unwilling to expose the frequency of changes to their back-end code to clients. Second, the above scheme would require the client to perform two remote attestations the first time it visits a website and one attestation with every subsequent visit—namely, with the target enclave—until the verifier enclave changes its long-term keypair. We observe that 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. Through this observation, we present a more efficient scheme that only requires the client to perform one *asynchronous* remote attestation that is valid as long as the corresponding server-side enclave keeps its long-term keypair. It does not require any further remote attestations per website visit, and never exposes the target enclave's enclave measurement to the client.

5.2 Mitigator's final design

In Mitigator, we offload this task of authenticating the target enclave to another enclave on the website provider, which we refer to as the *decryptor* enclave. We thus have three server-side enclaves in our design, which we depict in Figure 2. We have two stages following the verification stage. In the first such stage (the *deployment* stage), the decryptor enclave conducts local attestation with the verifier enclave in order to learn the expected signer measurement of the target enclave. It then performs *local* attestation with the target enclave and authenticates the latter using this measurement. The decryptor maintains the resulting secure channel with the target enclave for the third, *runtime* stage. (We remark that the three enclaves do not have to be co-located on the same server-side machine. They can be on different machines and in that case inter-enclave local attestation is replaced by remote attestation.)

In contrast to the strawman design, where the client performed remote attestation with the *target* and *verifier* enclaves, in the runtime stage of our final design the client performs remote attestation *only* with the *decryptor* enclave. All Mitigator-supporting websites share the same decryptor enclave code and it does not change with improvements in the verifier code (with better privacy policy model generators or information flow analysis tools) or the target website's code. Indeed, it is designed to change the most infrequently in comparison to the latter two codebases, and thus the client only needs to perform this attestation *whenever the decryptor's code changes*, rather than with every page load [20], or even when the *website's* code changes (strawman design). Therefore, although our design could include the ver-

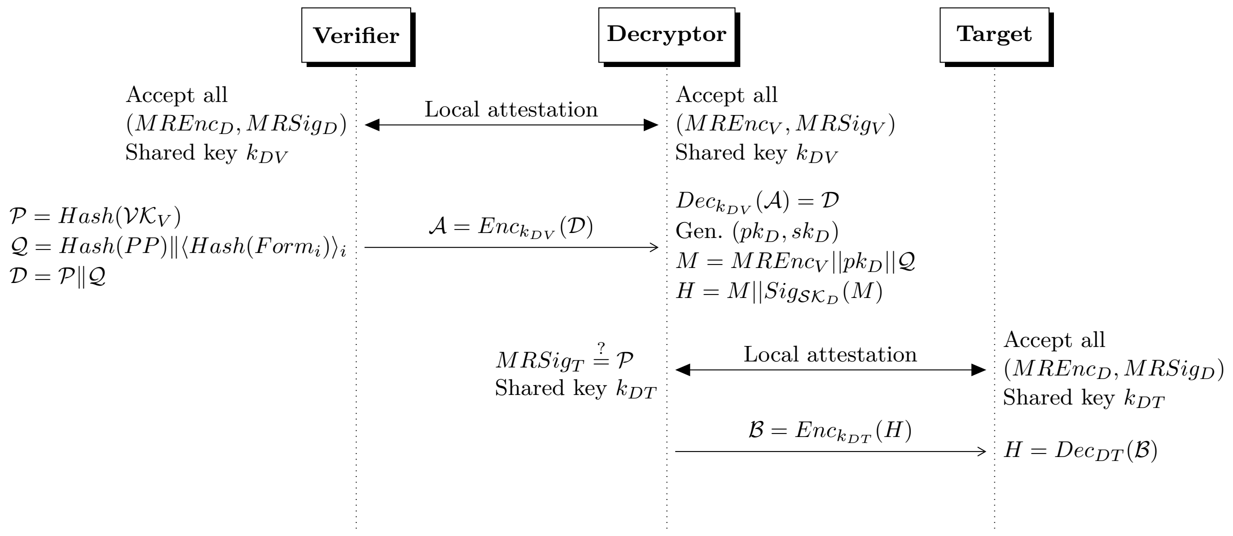
Verifier
Decryptor
Target
Accept all $(MREnc_D, MRSig_D)$ Shared key k_{DV}
Local attestation
Accept all $(MREnc_V, MRSig_V)$ Shared key k_{DV}
$\mathcal{P} = Hash(\mathcal{VK}_V)$
$\mathcal{Q} = Hash(PP) \| \langle Hash(Form_i) \rangle_i$
$\mathcal{D} = \mathcal{P} \| \mathcal{Q}$
$\mathcal{A} = Enc_{k_{DV}}(\mathcal{D})$
$Dec_{k_{DV}}(\mathcal{A}) = \mathcal{D}$
Gen. (pk_D, sk_D)
$M = MREnc_V || pk_D || \mathcal{Q}$
$H = M || Sig_{\mathcal{SK}_D}(M)$
$MRSig_T \overset{?}{=} \mathcal{P}$ Shared key k_{DT}
Local attestation
Accept all $(MREnc_D, MRSig_D)$ Shared key k_{DT}
$\mathcal{B} = Enc_{k_{DT}}(H)$
$H = Dec_{DT}(\mathcal{B})$

Fig. 3. Deployment stage. 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 decryptor enclave. (The decryptor enclave accepts all initiating enclaves.) The verifier sends the expected enclave measurement of the target enclave and $\mathcal{Q}$ (a hash of the privacy policy text and a hash of the structure of each compliant form on the site) to the decryptor enclave over the local attestation channel. The decryptor generates a token M that consists of the initiating enclave's enclave measurement, its own ephemeral public key, and $\mathcal{Q}$. It signs over it using its long-term signing key to obtain the token H. The target enclave then initiates a local attestation request to the decryptor enclave. The decryptor only accepts the target's request if the latter's signer measurement is the same as the one sent by the verifier earlier. If so, then the decryptor establishes a secure channel, which has the symmetric key k_{DT}, with the target. The decryptor then sends the token H over this channel to the target. These steps occur before any client connects to the website and enable the target enclave to forward the token H to the client.

ifier and the decryptor's functionalities within one enclave, our modular design supports auditability of both codebases and requires fewer remote attestations. Furthermore, the client only uses the secure channel established through remote attestation to obtain the decryptor's long-term verification key rather than to send users' data directly over it.

Following remote attestation, in the runtime stage, the client obtains a token, signed by the decryptor enclave, that includes the decryptor's current public encryption key, as well as the enclave measurement of the verifier enclave. The client ensures that this enclave measurement corresponds to that of a sound open-source verifier. It then encrypts form fields to the decryptor enclave's public key, which are submitted to the webserver running in the target enclave, which forwards them to the decryptor enclave over the aforementioned secure channel. Finally, as its name suggests, the decryptor enclave decrypts the form field data and returns it back to the target enclave over the secure channel. We detail each of these three stages below.

Verification stage. This stage only requires the verifier enclave to be online. We remind the reader that in this stage, the verifier enclave uses its long-term signing-verification keypair $(\mathcal{SK}_V, \mathcal{VK}_V)$ to sign over the website's source code and privacy policy, only if the former is compliant with the latter. We note here that the verifier enclave seals this keypair to disk using its enclave measurement, so that the untrusted OS, or any other enclave, cannot modify or access it.

Deployment stage. In this stage, all three enclaves need to be online. As above, the decryptor enclave generates and persists its long-term signing-verification keypair $\mathcal{SK}_D, \mathcal{VK}_D$ using its enclave measurement to seal it. Mitigator's client 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 for the next stage. After setting up its long-term signing-verification keypair and the above service, the decryptor enclave starts listening for local attestation requests. This stage involves the verifier enclave and then the target enclave conducting local attestation and secure communication with the decryptor enclave. A sequence diagram of messages sent and received in this stage is shown in Figure 3.

Importantly, we remark that neither the verifier nor the decryptor enclaves need to be signed with keys belonging to the client or any other party. As these enclaves will be executed on the website providers' machines, they can be signed using a keypair generated by the adversarial OS. We illustrate through our design that this does not diminish the security of our system.

Verifier ↔ decryptor: After signing the compliant target enclave, the verifier enclave conducts local attestation with the decryptor enclave and establishes a secure channel with it. (Although the decryptor does not check the enclave or signer measurement of the initiating enclave, it stores its enclave measurement, $MREnc_V$, and sends it to the client in the second part of this stage.) Over this channel, it sends the following values: the expected signer measurement over the source code files (i.e., a hash $Hash(\mathcal{VK}_V)$ of the verification key), and a list of hashes $\mathcal{Q}$ that consists of a hash of the privacy policy, and hashes of each verified form structure. (The *structure* of a form is a canonical representation of its DOM, excluding pre-filled field values.)

Upon receiving these values, the decryptor generates a short-term keypair (pk_D, sk_D) and a token M that consists of the verifier's enclave measurement $MREnc_V$, the list of hashes $\mathcal{Q}$, and the short-term public key pk_D. The decryptor enclave signs the token M using its long-term signing key $\mathcal{SK}_D$. The decryptor enclave then concatenates the token and the signature to form the token $H = M||Sig_{\mathcal{SK}_D}(M)$. The decryptor enclave then terminates the secure channel with the verifier enclave and waits for a second local attestation request. The token H will be sent to the client in the runtime stage. (The verifier needs to include the hash of the privacy policy and the hashes of the verified form structures to enable the client to detect an attack we discuss in the runtime stage description below.)

Target ↔ decryptor: The target enclave then 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, $hash(\mathcal{VK}_V)$, 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. This check ensures that the target enclave that obtains users' data in the runtime stage has been signed by the first enclave that performed local attestation with the decryptor enclave. When combined with the client's check in the runtime stage, that the latter enclave is indeed a valid verifier enclave, we obtain the composite guarantee that the users' data is only handed over to a compliant target enclave. In the final step, the decryptor enclave sends the token H under the aforementioned secure channel to the target enclave. The decryptor now waits for further data along the secure channel with the target enclave.

Runtime stage. 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 the response to 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 the decryptor enclave behind the website. It sends these encrypted fields to the target enclave in the place of the plaintext fields. The target enclave then sends these fields to the decryptor enclave over the secure channel established through local attestation. The decryptor enclave returns the plaintext form fields over the same channel. We next discuss communication between the client and the target enclave and subsequently discuss communication between the target and the decryptor enclave. We also illustrate the entire exchange of messages that occur in this stage in Figure 4.

Client ↔ target: The client performs three checks on the integrity of the signed token H as follows. First, 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 may have modified the token in transit. 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 list $\mathcal{Q}$ of hashes of the verified files (including the privacy policy and the form structures), 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 checked as sound. 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 without checking it. 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 linked on the website and the hash of the form structure, are contained in $\mathcal{Q}$. Again, if this check fails, then the adversarial OS may have served to the client a different privacy policy or a form with a different structure from the one that the verifier enclave

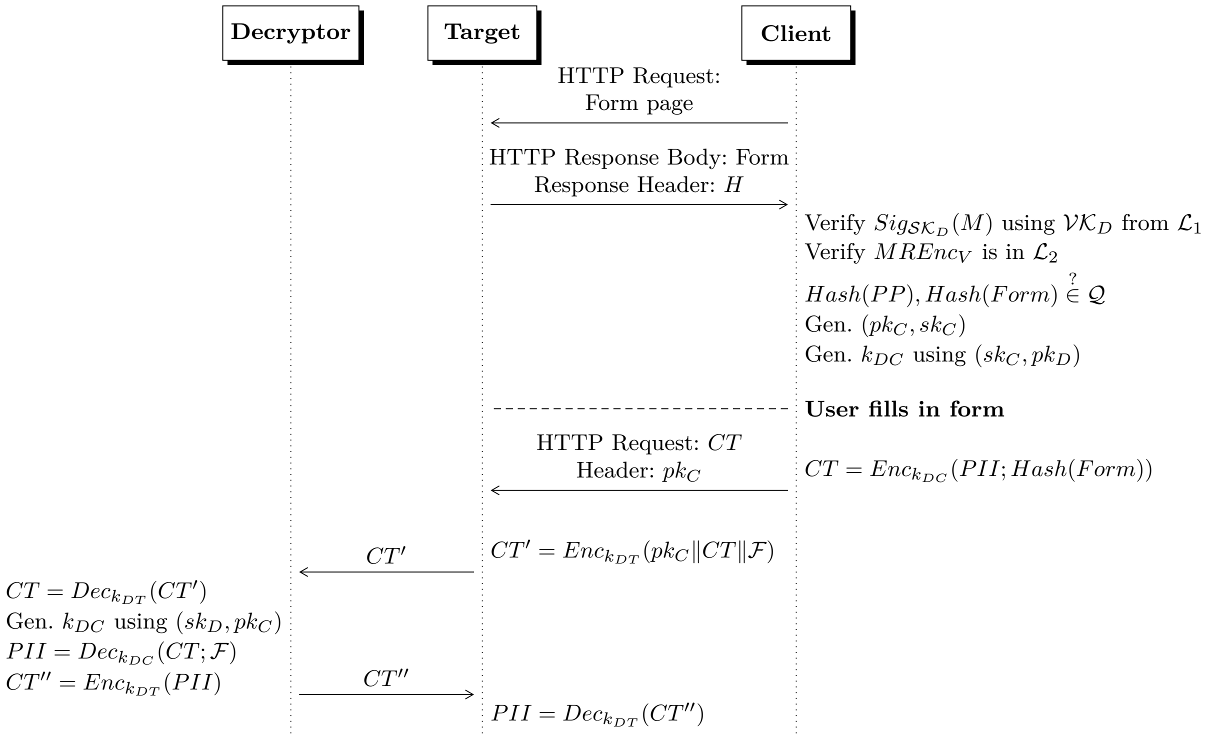
Decryptor
Target
Client
HTTP Request: Form page
HTTP Response Body: Form
Response Header: H
Verify $Sig_{SK_D}(M)$ using $\mathcal{VK}_D$ from $\mathcal{L}_1$
Verify $MREnc_V$ is in $\mathcal{L}_2$
$Hash(PP), Hash(Form) \overset{?}{\in} \mathcal{Q}$
Gen. (pk_C, sk_C)
Gen. k_{DC} using (sk_C, pk_D)
User fills in form
HTTP Request: CT
Header: pk_C
$CT = Enc_{k_{DC}}(PII; Hash(Form))$
CT'
$CT' = Enc_{k_{DT}}(pk_C \| CT \| \mathcal{F})$
$CT = Dec_{k_{DT}}(CT')$
Gen. k_{DC} using (sk_D, pk_C)
$PII = Dec_{k_{DC}}(CT; \mathcal{F})$
$CT'' = Enc_{k_{DT}}(PII)$
CT''
$PII = Dec_{k_{DT}}(CT'')$

Fig. 4. Runtime stage. Clients load a web page containing a form and receive a header containing the token H. 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 $\mathcal{L}_2$. It also computes hashes of the privacy policy and of the structure of the form page and checks they are in the signed token. If all checks are successful, it encrypts the user's form field data PII to the decryptor enclave, including the hash of the form structure it receives $Hash(Form)$ as the associated data in the encryption. It sends the ciphertext value CT and its public key pk_C to the target enclave instead of the plaintext value. The target enclave then forwards these requests, along with its own knowledge of the hash $\mathcal{F}$ of its form structure, to the decryptor enclave along the established secure channel with the symmetric key k_{DT}. The decryptor enclave uses the client's public key to derive its session key k_{DC} with the client, decrypts the ciphertext CT using $\mathcal{F}$ as the assoiated data, to obtain the plaintext value PII. It then re-encrypts this value back to the target enclave (CT'').

checked the source code against. (The adversarial OS may change the output HTML files shown to the client as the TLS connection may terminate outside the enclave.) 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. Therefore, to prevent this attack, the verifier sends $\mathcal{Q}$ to the decryptor enclave and the client performs this check.

If any of the three checks fails, then the client treats the website as if it did not support Mitigator and thus does not present the user with a positive signal of compliance. If all checks are successful, the client then signals to the user that the form has been successfully validated by Mitigator against the displayed privacy policy. The client then obtains the users' plaintext data and encrypts it to the decryptor enclave. The specific technique used to securely obtain the users' plaintext data through the browser may vary across implementations; we detail our specific strategy in Section 6.

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 6 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 third check, however, does not require any additional state to be maintained by the client. The client may simply compute the required hashes and check that they are in the signed token.

The client then generates its own short-term keypair (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}. Whenever a user clicks to submit the form, this key is used to encrypt the form field values, which may contain their personally identifiable information and are thus denoted by PII. This (AEAD) encryption also includes a hash of the form structure as associated data, so that a malicious OS cannot redirect the client's form submission to a *different*, albeit verified, form on the target server. The client sends the resulting ciphertext values CT 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 $\leftrightarrow$ decryptor: 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 obtains HTML form field data after decrypting with the TLS session key, as usual. However, for Mitigator-supporting websites, these HTML form fields are not plaintext data. They are ciphertexts (CT) that are encrypted to the decryptor's short-term public key pk_D. The target enclave then sends the client's pk_C, the client's ciphertext, and its own knowledge of the hash $\mathcal{F}$ of its own form structure, along the secure channel to the decryptor enclave and waits for a response.

Upon obtaining any data from the target enclave along the secure channel, the decryptor enclave removes the first (secure channel) layer of encryption, to obtain the ciphertext form fields sent by the client (CT), along with its short-term public key pk_C. The decryptor enclave 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 (AEAD) decrypts the ciphertext fields CT with $\mathcal{F}$ as the associated data to obtain the plaintext form fields PII. Finally, the decryptor enclave sends these plaintext fields back to the target enclave, along the secure channel. Again, upon obtaining any data from the decryptor enclave along the secure channel, the target enclave removes the secure channel encryption and obtains the plaintext form field data (PII).

5.3 Security analysis

As mentioned in Section 4, our threat model includes an adversarial OS that wishes to perform operations on users' data that are not permitted by 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 these attacks are either detected by Mitigator or are limited to the OS performing denial-of-service (DoS) attacks on its own Mitigator deployment. As we mentioned in Section 4, we assume that the website provider has incentives to stop DoS attacks against their own Mitigator setup.

While we do not trust the OS, we do trust the code running inside enclaves to function correctly and securely (without side channels, for example), as well as the enclave hardware itself. We therefore assume that any secrets generated by an enclave and stored in its state are not revealed to the adversarial OS. As the verifier's and decryptor's long-term signing keys are sealed using their enclave measurement as an input to the sealing algorithm, malicious verifiers or decryptors will not be able to unseal these keys to forge a signature as they will not have the needed enclave measurement. As a sound verifier would only sign over a compliant target enclave, the adversarial OS cannot obtain a valid signature over a noncompliant enclave by running a sound verifier. As all messages between the verifier and the decryptor enclaves are authenticated encryptions, the adversarial OS cannot fool the decryptor enclave into accepting a signer measurement of a noncompliant target enclave. Furthermore, an adversarial OS that attempts to run a lazy or unsound verifier that simply signs over any target enclave will be rejected by the Mitigator client in the runtime stage as the enclave measurement of such a verifier will be different from that of a genuine verifier.

We have discussed how an adversarial OS cannot spawn a noncompliant target enclave to obtain a client's plaintext data without being detected by the client. It could, however, attempt to obtain the client's ciphertext data before it reaches the target enclave, to modify or replay it. (This is even possible only if the TLS connection was terminated outside the enclave.) In case the adversarial OS modifies the client's public key or ciphertext values, the decryptor enclave will reject the modified ciphertexts. 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 since it last changed its encryption key pk_D and rejecting any repeated ciphertexts.

6 Implementation overview

In this section, we describe the components used to implement each of the three enclaves on an Intel SGX-supporting machine. We also discuss the implementation of Mitigator's client through a browser extension.

6.1 Decryptor enclave

The decryptor enclave includes code to interact with the verifier and target enclaves. We expect the complexity of its codebase to be independent of the verifier and target enclaves and we expect its codebase to not change much, unless the deployment or runtime stage protocols themselves change. We have thus developed it within the traditional SGX SDK setup, partitioning its code into an untrusted application ($\sim$1K hand-written LOC and $\sim$8K Protobuf auto-generated LOC) and a trusted enclave ($\sim$1K LOC). We reiterate that our client-side implementation only needs to conduct remote attestation with the decryptor enclave when the decryptor's source code changes.

6.2 Verifier enclave and target enclaves — Graphene-SGX

The target and the verifier both consist of running complex native applications; namely, a webserver application and a source code analysis tool adapted to perform a compliance check for a simple policy model. For this reason, Mitigator's verifier and target enclave programs are run on top of a shim library, named Graphene-SGX [9], in order to seamlessly support system calls within the unmodified applications, allowing them to run within an SGX enclave. We remark that other approaches, such as Shinde et al.'s Panoply [26] and Arnautov et al.'s Scone [3], also address the problem of running general-purpose applications within trusted hardware platforms. However, as the Graphene-SGX implementation supported easily running unmodified applications within enclaves, we chose to use Graphene-SGX.

Graphene-SGX allows applications' dynamic libraries and configuration files to be included in the enclave measurement, and specified in a manifest file. At runtime, Graphene's dynamic loader prohibits any such libraries or files whose hash is not present in the manifest file. Thereby, it supports applications that are linked against dynamic libraries or that read configuration or other files. We extended Graphene-SGX to support attestation and sealing by including portions of the corresponding SGX SDK libraries.

Verifier enclave. We proceed to discuss the verifier enclave implementation. As mentioned previously, we utilize Pixy within Mitigator. In particular, 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. Any plaintext data that the webserver obtains from the decryptor enclave is tracked to ensure that the above policy holds; that is, such plaintext data should not be written to disk. Our implementation currently uses Pixy's configuration setup for listing unacceptable output functions for this data as a simplified privacy policy model. The simplified model is not a limitation of our design; as mentioned previously, other tools like Harkous et al.'s Polisis [16] or Andow et al.'s PolicyLint [2] could be used to automatically extract relevant models that can be used to configure source code analysis tools for compliance checking.

Target enclave. Graphene-SGX allows us to run a native, unmodified webserver within an Intel SGX enclave. We use Graphene-SGX to run an Apache webserver (serving PHP pages) and a PHP extension that implements the runtime stage interactions with the client. Mitigator's PHP extension is responsible for three tasks. First, it should perform local attestation with the decryptor enclave once when the server is set up to establish a secure channel with that enclave. Second, 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. This token H is then sent by the calling PHP script to the client through a custom header. Finally, Mitigator's client implementation would encrypt form field data to the decryptor enclave and send in the client's public key pk_C through another custom header. The PHP extension should enable any calling PHP scripts to pass this key, the client's ciphertext data, and its form structure hash, and expect plaintext client form fields in response. Mitigator's PHP extension implements these three functionalities; our source code can be accessed at https://git-crysp.uwaterloo.ca/miti/mitigator.

6.3 Browser extension

Mitigator's browser extension implements a Mitigator client's functionalities. First, the browser extension is responsible for sending and receiving Mitigator-specific tokens to and from the PHP extension through custom headers. Second, it verifies the integrity of the server-side Mitigator token that it receives by performing the three checks outlined in Section 5.2. It uses a list of long-term decryptor verification keys for the decryptor enclave behind each Mitigator-supporting URL ($\mathcal{L}_1$) and a list of expected enclave measurements for sound verifiers ($\mathcal{L}_2$). List $\mathcal{L}_1$ can either be maintained by the client itself by performing remote attestations with each site's decryptor enclave, or it can trust some entity to maintain this list of remotely attested decryptor enclave public keys. This list need only change when new sites add support for Mitigator, when the decryptor code itself (very infrequently) changes, or when existing sites change the physical hardware used to run the decryptor enclave. (Even if Mitigator were extended to a load-balanced setting where each server machine has a target and decryptor enclave, then the client may be required to conduct a synchronous RA with a decryptor enclave if it encounters one behind a new server. In any case, the number of client-decryptor RAs scales with the number of different servers, rather than the number of page loads done by the client.)

List $\mathcal{L}_2$ is maintained by an entity that is trusted to conduct audits of the *verifier* source code, to ensure its soundness. This entity could be the same or different as the one above; it could, for instance, simply be a decentralized group of open-source developers. Importantly, this entity never needs to see the source code of the *target* enclave—the website business logic—itself, and this list need only be updated when improvements to the verifier program are made, not when the website code is updated. Finally, the maintainers of $\mathcal{L}_1$ and $\mathcal{L}_2$ are not involved in the runtime stage. For these reasons, they are not a single point of failure for processing Mitigator-protected forms and do not ever learn what websites clients are visiting.

Finally, the browser extension should encrypt form field data to the site's decryptor enclave, and send the ciphertext instead of plaintext form field data. In our proof-of-concept implementation, the user simply enters their form field data into the webpage itself. When the user finishes typing their text, they can press the form submit button, as usual. This button click now results in the entered text fields to be encrypted to the decryptor enclave for the given website.

We remark that our current implementation, which takes in users' data from the website's original form, allows dynamic Javascript code to change the appearance or structure of the webpage. This leaves it vulnerable to user interface attacks similar to those pointed out by Freyberger et al. [14]. To avoid such attacks, a design such as that proposed by Krawcieka et al. [20] for SafeKeeper can be used instead, which only allows user-driven interaction with the browser extension's pop up to fill the form.

7 Evaluation

In order to gauge whether websites or users would actually want to deploy or use a Mitigator implementation, a detailed user study would be required. We leave such a study to future work. Even if users would like to use a Mitigator implementation, it is desirable that it should *not* incur a large overhead in comparison to an identical system without Mitigator's guarantees, 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 and deployment stages incur a one-time latency overhead; the latter only occurs once during the deployment process, and so we do not expect the latencies of these stages to hinder Mitigator's incorporation into an iterative software development lifecycle. We do not test latencies during these stages further as these latencies do not impact the runtime latency experienced by users.

7.1 Performance evaluation

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. The target enclave runs an Apache server executable within Graphene-SGX which links to the Mitigator PHP extension and serves three PHP files: the privacy policy file, the form page, and its action page. Once the user fills out the form page, they are directed to the action page, which simply uses the PHP extension for the decrypted, plaintext form field values, and prints them. This Apache server is known as the *Mitigator server*. 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; in practice, a deployment could choose whether to terminate TLS outside of the target enclave or within it, as in TaLoS [4].

Server-side computational latency. Our first experiment consists of measuring the server-side latency from a headless client on the same machine as the servers. This setup effectively measures the total *computational* latency as the network latency is very small. We modified the browser extension to generate n requests, each consisting of encryptions of random alphanumeric plaintext form fields of a given size to the decryptor enclave, and to send each request to each of the three servers. We vary the form field size in logarithmic increments and we thus have n requests for each form field size in our experiments. We also modify the extension to log these requests in order to test latencies from headless clients. We instrumented the Mitigator server's form action page to measure the wall clock time required for the target and decryptor enclaves. We also instrumented each of the latter enclaves to measure the wall clock time required to handle each request. We start by presenting the latencies we measured in this setup.

In Figure 5, we plot the averages and standard error bars for the following wall clock times, for $n = 50$ as the length of the single form field increases: the total network round-trip time for each of the three servers, the time spent within the PHP script for the Mitigator server, the time spent within the PHP extension, and the time spent solely within the decryptor enclave.

We draw the following conclusions from Figure 5:

1. The Mitigator server's RTT is significantly higher than that of the control server. However, 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 add a small computational overhead on top of that concomitant with running the Graphene-SGX shim library. It is evident, therefore, that our work would greatly benefit from advances in running unmodified applications within Intel SGX with minimal increase in latency.
2. Our computational overhead on top of the Graphene-SGX server includes the sum of the decryptor enclave latency and the PHP extension latency (the sum is shown in purple). Each of the latter is 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.

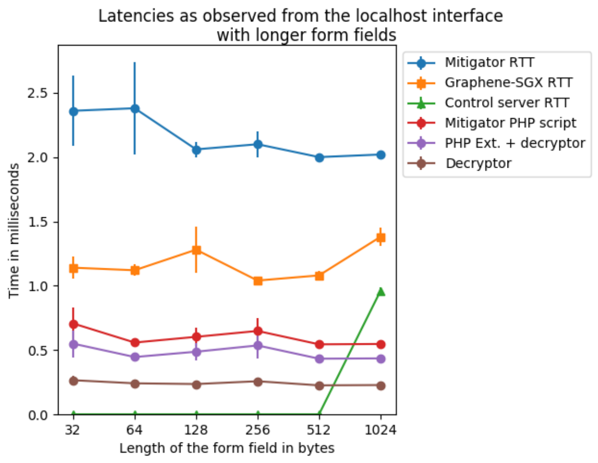

Fig. 5. Measurement of server-side latency from the localhost interface as the size of a form field increases: the length of a single form field was doubled in steps from 32 to 1024 and we measured the round-trip times (RTT) from the localhost interface for each of the Mitigator, Graphene-SGX, and control servers. This plot shows the average RTTs and in-enclave latencies over 50 requests for each form field length. Note that the plot in purple includes the sum of the plot in brown and the latency due to the PHP extension. Similarly, the plot in red includes the sum of both the extension and the decryptor enclave latencies as well as the PHP script latency.

3. 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.
4. 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.

Server-side network and computational latency. In light of the first observation 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 a machine on another network, which we refer to as *remote 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.) We plot the average in-enclave latencies and RTTs in Figure 6.

From Figure 6, we can observe that the total computation time within enclaves does not observably increase with the amount of PII data being encrypted,

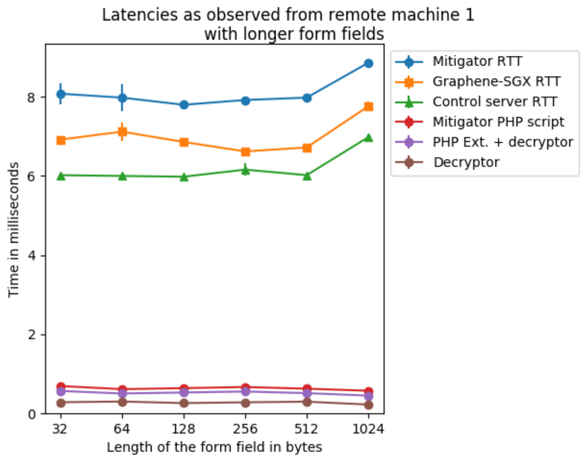

Fig. 6. An identical setup to that in Figure 5, with the exception that the round-trip times are measured from a headless client on another machine. The standard error bars for most values in this dataset are too small to be seen on this graph.

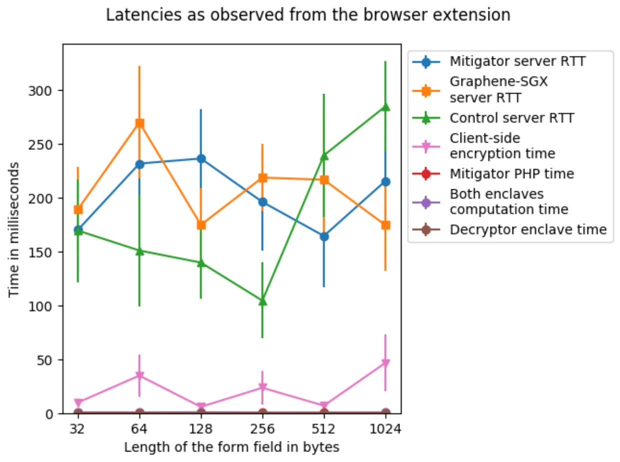

Fig. 7. An identical setup to that in Figure 5, with the exception that requests are sent from a browser extension, instead of a headless client, on another remote machine with higher network latency, and $n = 10$ requests are sent for each data point. The bottom three lines are all indistinguishable at the bottom of the graph.

and is much smaller than even a small network overhead, as is the case for a client on our remote machine. This shows that a server-side Mitigator implementation will be responsive and will incur a very small overhead with respect to the two aforementioned server implementations.

End-to-end latency. In the above experiments, 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 incurs additional latencies due to Mitigator's client-side components. We now proceed to measure the end-to-end latency of Mitigator as follows: we repeat the above experiment with the exception that the requests are also sent by the browser extension running on a network with higher latency and not a headless client. (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.) We instrument the browser extension to measure the wall clock time that it takes to encrypt a given message and the end-to-end network round-trip time for each of the three servers. We note that these network RTTs form a lower bound on what a user would expect to see as they exclude the time required to render the page; this time is very small in our case as the form action page only consists of HTML text. In Figure 7, we present the mean and standard errors for the latencies we obtained from sending $n = 10$ requests for each field size to each server.

Again, we find that the average server-side enclaves' latencies are negligible in comparison to network latencies, as measured from a browser extension on a remote machine. Second, the average encryption latency (shown in pink) remains less than 50 ms and is also smaller, by a factor of 2, than the average end-to-end latencies to any of the three servers. Finally, we remark that the average Mitigator server latencies are not discernibly different from those for the Graphene-SGX or control servers; the browser's internal processes add add so much noise as to completely mask the small additional latency added by Mitigator. We can thus safely conclude that the Mitigator implementation's end-to-end latency is largely impacted by the network latency, rather than server-side or client-side computational latency and we have grounds to believe that users may not perceive a Mitigator implementation to be adding a significant latency to their normal browsing experience.

8 Future work

There are many aspects of our design and implementation that can can be improved upon in future work. First, integrating a privacy policy model generator with a source code analysis tool is essential and challenging for implementing a verifier for any non-trivial website codebase and privacy policy. 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 back-end source code, such as the system currently being developed by Zimmeck et al. [34] for smartphone apps and policies. Extending our 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.

Mitigator does not provide users guarantees over client-side code, however, modern websites use dynamic Javascript code to process users' data on their machines. Assuring compliance of client-side code remains a very relevant adjacent problem, especially as users may reasonably have a mental model that expects a Mitigator implementation to provide its guarantees over such dynamic, client-side code as well. Last but not least, conducting a thorough usability study to determine whether different users, developers, and website owners *want* to use or support a Mitigator implementation, and if so, if it is *usable* or user-friendly, is thus a relevant line of future work.

9 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 have provided a design called Mitigator, and outlined a proof-of-concept implementation for it. Additionally, we have evaluated the performance of our system through latency analysis and have found that its end-to-end latency is largely impacted by network latency rather than server-side or client-side computational latency. We hope that Mitigator opens up further opportunities for research and development of prototypes to enable users to be assured that source code that processes their data is compliant with the written claims that are provided to them.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. We also thank Chia-Che Tsai and Donald E. Porter for publishing their code for Graphene-SGX online. We gratefully acknowledge the Royal Bank of Canada and NSERC grant CRDPJ-534381 for funding this research. This research was undertaken, in part, thanks to funding from the Canada Research Chairs program. This work benefitted from the use of the CrySP RIPPLE Facility at the University of Waterloo.

References

[1] 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.

[2] Benjamin Andow, Samin Yaseer Mahmud, Wenyu Wang, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Tao Xie. PolicyLint: Investigating Internal Privacy Policy Contradictions on Google Play. In *28th USENIX Security Symposium*, pages 585–602, Santa Clara, CA, August 2019. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity19/presentation/andow.

[3] 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)*, pages 689–703, Savannah, GA, 2016. USENIX Association. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

[4] 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. URL https://www.doc.ic.ac.uk/research/technicalreports/2017/DTRS17-5.pdf.

[5] 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)*, pages 334–349, 2017. 10.1109/EuroSP.2017.14.

[6] 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, pages 155–167, New York, NY, USA, 2018. ACM. 10.1145/3267323.3268954. URL https://dl.acm.org/citation.cfm?id=3268954.

[7] 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)*, pages 163–172, 2014. 10.1109/RE.2014.6912258.

[8] 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)*, pages 1041–1056, Vancouver, BC, 2017. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck.

[9] 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)*, pages 645–658, Santa Clara, CA,

2017. USENIX Association. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

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

[11] Lorrie Cranor, Marc Langheinrich, Massimo Marchiori, Martin Presler-Marshall, and Joseph Reagle. The Platform for Privacy Preferences 1.0 (P3P 1.0) Specification. https://www.w3.org/TR/P3P/, 2002.

[12] Lorrie Faith Cranor. Necessary but not sufficient: Standardized mechanisms for privacy notice and choice. *Journal on Telecommunications and High Technology Law*, 10:273–308, 2012. http://jthtl.org/content/articles/V10I2/JTHTLv10i2_Cranor.PDF.

[13] 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)*, pages 637–654, Austin, TX, 2016. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/elnikety.

[14] 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. *Proceedings on Privacy Enhancing Technologies*, 2018(2):47–63, 2018. http://dx.doi.org/10.1515/popets-2018-0012.

[15] 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)*, pages 47–60, Hollywood, CA, 2012. USENIX. URL https://www.usenix.org/conference/osdi12/technical-sessions/presentation/giffin.

[16] 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)*, pages 531–548, Baltimore, MD, 2018. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity18/presentation/harkous.

[17] Intel Corporation. Intel® Software Guard Extensions (Intel® SGX) Developer Guide. https://download.01.org/intel-sgx/linux-2.5/docs/Intel_SGX_Developer_Guide.pdf, 2019.

[18] 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. 10.1109/SP.2006.29.

[19] Jayanthkumar Kannan, Petros Maniatis, and Byung-Gon Chun. Secure Data Preservers For Web Services. In *2nd USENIX Conference on Web Application Development*, WebApps'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2002168.2002171.

[20] 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, pages 349–358, Republic and Canton of Geneva, Switzerland, 2018. International World Wide Web Conferences Steering Committee. 10.1145/3178876.3186101.

[21] 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. https://keystone-enclave.org/, 2019.

[22] 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)*, pages 557–574, Vancouver, BC, 2017. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho.

[23] 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*, volume 7, pages 193–205. USENIX Association, 2011. URL https://www.usenix.org/legacy/event/hotos/tech/final_files/ManiatisAkhawe.pdf.

[24] 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, 2018. URL http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_02B-4_Sasy_paper.pdf.

[25] 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, pages 327–342, Washington, DC, USA, 2014. IEEE Computer Society. 10.1109/SP.2014.28.

[26] 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, 2017. 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/.

[27] 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*, pages 25–36. ACM, 2016. 10.1145/2884781.2884855.

[28] 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*, pages 991–1008. USENIX Association, 2018. URL https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[29] Frank Wang, Ronny Ko, and James Mickens. Riverbed: Enforcing user-defined privacy constraints in distributed web services. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 615–

630, Boston, MA, 2019. USENIX Association. ISBN 978-1-931971-49-2. URL https://www.usenix.org/conference/nsdi19/presentation/wang-frank.

[30] Zack Whittaker. Equifax breach was 'entirely preventable' had it used basic security measures, says House report. https://techcrunch.com/2018/12/10/equifax-breach-preventable-house-oversight-report/, 2019.

[31] 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, 2016. ACL. dx.doi.org/10.18653/v1/P16-1126. URL https://www.aclweb.org/anthology/P16-1126.

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

[33] 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, 2017. Internet Society. URL https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/automated-analysis-privacy-requirements-mobile-apps/.

[34] Sebastian Zimmeck, Peter Story, Rafael Goldstein, David Baraka, Shaoyan Li, Yuanyuan Feng, and Norman Sadeh. Compliance Traceability: Privacy Policies as Software Development Artifacts. Open Day for Privacy, Usability, and Transparency, July 2019. https://sebastianzimmeck.de/zimmeckEtAlTraceability2019Abstract.pdf.