Using Dynamic Software Product Lines to Implement Adaptive SGX-enabled Systems

Sebastian Krieter
Harz University of Applied Sciences
Wernigerode, Germany
Otto-von-Guericke-University
Magdeburg, Germany

Tobias Thiem
Harz University of Applied Sciences
Wernigerode, Germany

Thomas Leich
Harz University of Applied Sciences
Wernigerode, Germany
METOP GmbH
Magdeburg, Germany

ABSTRACT

In the light of computational outsourcing and external data storage, data protection and trusted execution become increasingly important. Novel hardware such as Intel's Software Guard extensions (SGX) attempts to provide a solution to protect data and computations from unauthorized access and manipulation, even against attackers with physical access to a machine. However, the current generation of SGX limits the protected memory space that can be efficiently used to 128 MiB, which must be shared between data and binary code. Thus, we propose to use a software product line approach to tailor an application's binary code in such a way that it can be updated during runtime, with the goal to only store relevant features in the protected memory at a given time. We provide a prototypical implementation that enables basic support for loading and unloading features during runtime and evaluate our prototype in terms of execution times against non-adaptive execution.

CCS CONCEPTS

• **Software and its engineering → Software product lines**.

KEYWORDS

Software Product Lines, Intel Software Guard Extensions, Runtime Adaptation

ACM Reference Format:
Sebastian Krieter, Tobias Thiem, and Thomas Leich. 2019. Using Dynamic Software Product Lines to Implement Adaptive SGX-enabled Systems. In *Proceedings of 13th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS '19).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3302333.3302340

1 INTRODUCTION

Handling data-intensive computations can be a costly and error-prone endeavor for many companies, as they need to acquire, run, and maintain the necessary hard- and software, which may not be their expertise. Thus, many companies try to reduce their cost and risk by relying on third-party service providers to outsource their data storage, data processing, or both [13]. The outsourcing process can range from simply storing data externally to using specialized services or even running software on externally hosted infrastructure [4]. Either way, whenever sensitive data is stored on external machines, data owners have to trust the third-party service provider and bear the risk of data leaks or data loss [8, 25].

A possible solution to this issue is the usage of modern hardware for encryption and trusted execution such as Intel's Software Guard Extensions (SGX) [15]. Using hardware solutions can shift the trust from service providers to hardware manufacturers, as it ensures that sensitive data and computations cannot be read or manipulated by unauthorized entities even with physical access to a machine. In this paper, we focus on SGX as one particular hardware solution for trusted execution. With SGX, Intel equips some of their CPUs with an extended instruction set to isolate parts of the main memory [21]. However, using SGX adds new challenges to develop an application.

Currently, the most critical problem is the space limitation of SGX-secured main memory, the enclave page cache (EPC). To efficiently process data, both the data and the application's binary code must reside inside the EPC. However, on current hardware, the EPC is limited to 128 MiB. All data that does not fit inside it is paged out to the unprotected memory and thereby causes massive performance degradation. Thus, it is desirable to only store code and data inside the EPC that is needed for the current task at hand.

In this paper, we propose an approach that aims to reduce the size of an application's binary code, to increase the amount of data it can fit inside the EPC. Our goal is to decompose applications into a set of features and enable loading and unloading of these features into the EPC during runtime. Thus, we present a novel approach for dynamically adapting SGX-enabled systems using techniques from software product line engineering (SPLE). With this approach, we take a step towards our vision of a secure and scalable platform for applications inside the cloud [16].

2 BACKGROUND

In the following, we briefly explain the functionalities of Intel's Software Guard Extensions (SGX). We also define our notion of features, feature models, configurations, and dynamic software product lines. Furthermore, we provide relevant information on the Unix Executable and Linking Format (ELF), as our variability mechanism requires binary code in this format.

2.1 Intel SGX

The Intel Software Guard Extension (SGX) is a set of instructions for certain Intel CPUs with the purpose of enabling trusted computing in untrusted environments [2, 21]. With these instructions developers are empowered to transparently encrypt parts of the

VAMOS '19, February 6–8, 2019, Leuven, Belgium
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6648-9/19/02…$15.00
https://doi.org/10.1145/3302333.3302340

main memory, in which they can store sensitive data and securely execute their application code.

SGX was designed for an environment, in which an application runs on an external system. Both, the hardware and higher privileged software (e.g., the operating system) are controlled by a third party, who is not trustworthy. An attacker may read and manipulate arbitrary parts of the memory. Thus, the goal of SGX is to provide integrity and confidentiality for sensitive data and computations by forbidding unauthorized reads and detecting manipulations.

2.1.1 Enclaves. SGX protects data by providing secure containers called *enclaves*, which are the central components of this technology [10]. An enclave is protected from unauthorized access and contains all code and data that is necessary for running a computational task. Whether a given application runs inside an enclave can be verified from other machines by using remote attestation.

As an example, consider a database application on a remote server. A typical use case is that a user wants to retrieve some data from the database by sending a specific query. As the user does not trust the server provider, the database application runs inside an enclave. In this scenario, the enclave has three distinct tasks: (1) decrypt and verify the query, (2) execute the query, and (3) encrypt and sign the result. The remaining work (e.g., client-server communication) can be done outside the enclave as the handled data can neither be read nor manipulated due to the employed cryptographic methods. In turn, the user must also encrypt and sign the query and decrypt and verify the returned result.

Enclaves are enabled by isolating a part from the main memory called the *processor reserved memory* (PRM), which contains the *enclave page cache* (EPC), which, in turn, contains one or more enclaves [10]. The PRM is protected by the CPU against access from other processes and peripherals.

In order to securely execute some task, there are always two contexts for an application, an untrusted and a trusted part. The binary code of the trusted part resides inside one or more enclaves, whereas the untrusted part's binary code is stored in the regular memory. At program start, the enclave is securely stored (i.e., encrypted and signed) in the unprotected memory. At some point during the execution the enclave is loaded into the protected memory and initialized using the corresponding CPU instructions. This must be done in the untrusted part of the application. The untrusted part then may verify whether the enclave is indeed running inside the protected memory using remote attestation. When an enclave is initialized correctly the execution context can switch to run code inside the enclave. As soon as the execution of the trusted part is finished, the enclave can be destroyed from within the untrusted part to free space in the EPC.

There are two functions for switching the memory context. These are called *ecalls* and *ocalls*. An ecall enters an enclave by calling a method inside of it. Parameters of that method are copied to the protected memory and the result is written back to the regular memory. An ocall leaves the enclave and calls a method on the outside. The execution continues within the application in the untrusted part. Each of these transitions are relatively costly and, thus, should be used only if necessary, to avoid performance decrease.

All code inside an enclave runs within the user space and only has user privileges. That means that is not possible to make system calls inside an enclave (e.g., reading or writing to the file system), as they require higher privileges. In order to make a system call, the application must leave the enclave with an *ocall*, execute the corresponding command, and then return the result.

2.1.2 Enclave Page Cache and Paging. Currently, one of the main issues with SGX is the limited size of the PRM, which is 128 MiB on the most recent hardware. In order to process data and code inside an enclave, the enclave must be available inside the EPC. However, some enclaves may need to store more data or code than fits inside the EPC. This problem can be solved by using paging. An enclave is split into single 4 KB pages and, thus, the EPC only needs to hold pages, containing the momentarily required code and data. If an enclave is too large to fit inside the EPC, it is paged to the unprotected memory space. The downside of this mechanism is the performance impact that occurs whenever a page must be written to or read from the unprotected memory. In addition to copying pages between both memory regions, the SGX driver must use cryptographic functions to ensure that pages in the unprotected memory stay confidential and loaded pages were not modified outside the enclave.

Paging is handled by the particular SGX driver on the system. Therefore, the application normally has no control over which pages of an enclave are loaded and which ones are paged out. As the SGX driver does not know an application's context, there may arise situations, in which an application constantly requires data from a page that is paged out at the moment, leading to massive performance degradation.

2.2 Variability Management

We want to address the paging problem of the SGX driver by encapsulating binary code within features rather than single pages. Thus, making the memory management context-aware and ultimately improving caching behavior. In order to enable such an adaptive handling of binary code, we rely on the variability management of dynamic software product lines. In our context, a product is the trusted part of an SGX-enabled application that comprises a subset of the functionality defined by the feature model. In the following, we briefly describe our definition of features, feature models, configurations, and variability mechanisms.

2.2.1 Feature Model. In the context of this paper, we define a feature as a distinct functionality of a software product line [3]. A feature model defines the set of features of a software product line and their interdependencies [3, 6, 11]. In Figure 1, we present a small example of a feature model displayed as a feature diagram. In this diagram, each feature is represented by a corresponding node in a tree [12]. The structure, edge types, and group types of the tree model the features' dependencies.

2.2.2 Configuration. A configuration represents a product of a product line by specifying the set of selected features [3]. We call a configuration *valid*, iff it satisfies all dependencies defined by the feature model. An example of a valid configuration for the feature model given in Figure 1 is $c_1 = \{Calculator, Function, Add\}$, which selects the respective features and deselects the remaining ones (i.e., *Verbose* and *Mult*).

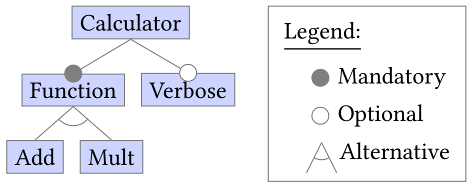

Figure 1: Example SPL of a simple calculator.

2.2.3 Variability Mechanism. In order to generate a working product from a software product line, a variability mechanism has to implement the variability defined by a valid configuration. The variability mechanism reuses the defined features to implement a product that behaves according to the given configuration. There exist many different variability mechanism, for instance feature-oriented composition, preprocessors, and plug-in systems [3].

As we aim to add and remove features from an enclave, while being executed, we are interested in a variability mechanism that can update a configuration at runtime. Such a mechanism is necessary to enable the implementation of dynamic software product lines [14, 22]. Nevertheless, since we have to consider the restrictions from the SGX environment, in this paper, we present an own variability mechanism (cf. Section 3.3), which is a dynamic approach that works on a granularity level of methods.

2.3 Executable and Linking Format

Our variability mechanism relies on manipulating the binary data inside an enclave. To this end, we need to extract the binary code of a method from compiled source files (i.e., binary files). We developed our approach for Linux and, thus, consider the corresponding binary format. The *Executable and Linking Format* (ELF) is the standard binary format of Unix-like systems [9].

An ELF file may serve different purposes. In general, there are three main types, of which we consider the relocatable file. A relocatable ELF file holds the compiled, but not yet linked source of an application. In addition, there are two views on an ELF file, the linking and the executable view. In this paper, we focus on the linking view, which consists of the ELF header, a set of sections, and the section header table.

The ELF header contains general information about file, such as the ELF type and the offset and length of the section header table. Each section is one continuous, non-overlapping part of the ELF file that holds some kind of data. To locate and identify sections, the section header table holds the relevant information about each section in the ELF file, such as its name, type, offset, and length.

For our approach, we consider the sections `.sysmtab` and `.text`. The `.text` section contains the executable binary code of the application. The `.sysmtab` section represent a table containing all symbols of an application (e.g., method names). In the case of a method symbol, the table also contains the relative address of the function inside the `.text` section.

3 DYNAMIC FEATURE MANAGEMENT

Employing Intel's SGX, we intend to enable trusted execution for an arbitrary application. Ideally, the entire binary code of a given application is securely executed within an enclave, while the only operations run from the untrusted memory are initializing and entering the enclave. However, as the memory protected by SGX is limited in size, we propose to modularize the application into a dynamic software product line, in order to be able to configure the enclave's binary code at runtime.

With our approach, we intend to enable dynamic adaption of code inside an enclave by allowing to load and unload features from the product line. Ultimately, we aim to reduce the amount of unnecessary code within the EPC to a minimum, which leaves more space for data or other enclaves running on the same system. To this end, we assume that often only a certain part of the application's entire functionality is needed at a given time.

Our approach consists of three parts, the *modularization* of the application to split the source code into distinct features, the *bit extractor*, which extracts the binary data for each feature from the compiled application, and the *variability manager*, which is responsible for loading and unloading the features into the enclave during runtime. The general workflow of the development is the following:

1. Before compile time (Modularization)
   a) Add annotations
   b) Model dependencies
   c) Modifying method calls
2. During compile time (Bit Extractor)
   a) Compile and link static and core features
   b) Compile optional features
   c) Use bit extractor to create feature files
3. During runtime (Variability Manager)
   a) Load enclave with static and core features
   b) Load optional features

In the following, we describe all parts of our approach in more detail.

3.1 Modularization

As a first step in our approach, we decompose an application's source code into features on method level. This means each method can be assigned to a feature by adding annotations to the code. This is a manual process, which has to be done by the developer before compiling the source. We also need to modify certain methods calls and specify all dependencies within the feature model.

3.1.1 Add Annotations. The mapping of a method to a feature is done by adding an annotation above the method. An annotation has the form of a line comment with the syntax `//@feature <feature_name>`. A feature can be mapped to one or more methods, but each method may only belong to one feature at maximum. We use annotations, because they are compatible with the C standard syntax, can be parsed relatively easily, and can be added to existing source code without much effort or restructuring.

In Listing 1, we show an example for a small calculator application written in C. It consist of three user-defined methods, `add` (cf. Line 6), `mult` (cf. Line 12), and `main` (cf. Line 17). We used an annotation for the feature *add* to the method `add` (cf. Line 5) and an annotation for the feature *mult* to the method `mult` (cf. Line 11).

In the resulting variability, we differentiate between three types of features within the source code, namely *static*, *core*, and *optional* features. As static features, we consider all code that is provided

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4
 5 //@feature add
 6 static int add(int a, int b)
 7 {
 8     return a + b;
 9 }
10
11 //@feature mult
12 static int mult(int a, int b)
13 {
14     return a * b;
15 }
16
17 void main(int argc, char *argv[])
18 {
19     if (argc == 4)
20     {
21         int a = atoi(argv[2]);
22         int b = atoi(argv[3]);
23         if (strcmp(argv[1], "add") == 0)
24             printf("Sum is %i", add(a, b));
25         else if (strcmp(argv[1], "mult") == 0)
26             printf("Product is %i", mult(a, b));
27     }
28 }
```

Listing 1: Simple C program with two features.

Table 1: Method calls to other types of features.

From \ To	Static	Core Feature	Feature A	Feature B
Static	Direct	No	No	No
Core Feature	Direct	Direct	Indirect	Indirect
Feature A	Indirect	Indirect	Direct	Indirect
Feature B	Indirect	Indirect	Indirect	Direct

```
 1 void main(int argc, char *argv[])
 2 {
 3     if (argc == 4)
 4     {
 5         int a = atoi(argv[2]);
 6         int b = atoi(argv[3]);
 7         if (strcmp(argv[1], "add") == 0)
 8         {
 9             oprintf("Sum is %d", (int)
                   getMethod("add")(a, b));
10         }
11         else if (strcmp(argv[1], "mult") == 0)
12         {
13             oprintf("Product is %d", (int)
                   getMethod("mult")(a, b));
14         }
15     }
16 }
```

Listing 2: Main method with modified method calls.

by statically linked methods such as the C standard library. Core features contain user-defined code that is always available during runtime (i.e., it cannot not be unloaded). Both, static and core feature are loaded immediately when initializing the enclave. The remaining user-defined code is distributed among optional features, which can be can be loaded and unloaded into/from the enclave during runtime. Only optional code must by marked with annotations, in order to map it to a certain feature.

Regarding our example in Listing 1, the method `main` belongs to the core feature. In contrast, the methods `add` and `mult` are optional and belong to the features *add* and *mult*, respectively. Additionally, the code contains calls to some statically linked methods, such as `strcmp` (cf. Line 23) and `printf` (cf. Line 24).

3.1.2 Model dependencies. When decomposing the application into its single features, the developers must model their interdependencies. It is up to the developers to decide what they consider to be meaningful features and how to structure their product line. From a technical point of view, a meaningful feature would comprise a single, reusable functionality. Thus, minimizing the dependencies between features and, by this, maximizing the possibilities to unload features. However, as this is hardly realizable in practice, in this paper, we do not focus on how to achieve a reasonable decomposed application in the context of SGX-enabled systems. We consider this to be out of scope and a topic for future work. Rather, we assume that an application already consists of multiple optional features provided by the developer.

3.1.3 Modifying Method Calls. As we intend to dynamically load features during runtime, we cannot know the memory location of their binary code during compilation. Thus, while we can link method calls within the same feature using relative memory addresses, we are not able to statically link calls to methods from other features. To solve this problem, we need to modify certain method calls within the source code by replacing them with a call to the variability manager. That is, instead of calling a method directly, we request a function pointer from the variability manager for the specific method using `getMethod` (cf. Section 3.3) and then use the pointer to call it indirectly. This allows us to avoid static linking between different features. In turn, with this approach, we introduce a performance overhead (cf. Section 5) and can no longer guaranty type safety (i.e., developers must make sure that they correctly pass parameters and cast return values).

In Table 1, we summarize, whether and how it is possible to call a method from a different type of feature. From within static features it is not possible to call any other feature method at all. A core feature can directly call its own methods and all methods from static features without the need to change the source or binary code. It is possible within a core feature to call a method from an optional feature (e.g., call `add` from within the main method (cf. Line 24)). However, as these optional features are not present during compile time, all method calls must be replaced in the source code of the core feature with calls to the variability manager. Regarding optional features, a feature can always directly call its own methods. However, all other calls must be replaced in the source code with a call to the variability manager.

We modify the method `main` from our example in Listing 1 by replacing the calls to the feature methods `add` and `mult` as these belong to optional features. In Listing 2, we display the result of this transformation (cf. Line 9 and Line 13).

In addition to replacing calls to other features, the application itself must also be prepared for the execution within the protected memory. As it is not possible the make system calls, some methods must be replaced with corresponding methods from the SGX trusted libraries or, in case they cannot be replaced, wrapped within an ocall instead. For instance, in Listing 2, we replaced all occurrences of the method `printf` with `oprintf` (e.g., Line 9), which wraps the call to `printf` inside an ocall.

3.2 Bit Extractor

The bit extractor is our tool for retrieving the binary code for each feature. It takes the compiled binary files and source code of the features as input and computes a file for each feature in our feature format, which can be parsed by the variability manager. In the following, we describe the build process of static, core, and optional features, the extraction process, and the resulting feature format.

3.2.1 Build Features. Both static and core features can be compiled using the SGX tool chain for compiling sources. This results in the signed enclave binary and the binary for the untrusted part of the application.

Regarding optional features, their source can be compiled normally using a regular compiler such as the GNU Compiler Collection (gcc). However, the linking to other features is skipped, because this is done through the variability manager during runtime. The resulting ELF files can be processed with our bit extractor tool.

3.2.2 Extracting. The input for our bit extractor tool consists of the source of a feature and its compiled binary files. From the annotations in the source code it infers the mapping of each method to its feature. To extract the binary code of each feature, the bit extractors locates the address of each method from the `.symtab` section of the given ELF files and then retrieves the respective code from the `.text` section. It then saves this information within a feature file in using our feature format.

3.2.3 Feature Format. In order to prepare the loading of features at runtime, we must store the corresponding binary code in a way that is easily accessible for the variability manager. For each feature, we create a file that consists of a list of exported method names for this feature and the corresponding binary data from the ELF file.

In Table 2, we show the structure of a feature file. At the top of the file there are two integers, representing the number of methods within this file and the length of the binary code in bytes. The following bytes represent the list of methods. Each entry in the list consists of an integer that states the length of the method name in bytes, an integer that indicates the relative position within the binary code, and a series of bytes that represent the method name. As the entries are of variable length, because of the method names, a feature file must be read in a linear fashion and, thus, parsed completely when loading a feature.

Regarding our example in Listing 1, the bit extractor creates two files, `add.f` and `mult.f`, for each feature, respectively. In the feature file `add.f` the list of methods contains only `add`, followed by the corresponding binary code. We display the contents of this file in a hexadecimal representation in Figure 2. Analogous, `mult.f` contains the method *mult* and its corresponding binary code.

Table 2: Feature Format Structure

Content	Data Type
Number of methods	`uint32_t`
Length of binary code	`uint32_t`
List of methods	
Length of method name	`uint32_t`
Offset within binary code	`uint32_t`
Method name	`char *`
Binary code of all methods	`char *`

```
add.f
    01000000                  - number of methods (1)
    14000000                  - code length       (20)
    00000000                  - offset            (0)
    03000000                  - name length       (3)
    616464                    - method name       (add)
    554889e5897dfc8975f8      - code
    8b55fc8b45f801d05dc3
```

Figure 2: Hexadecimal representation of the file `add.f`.

3.3 Variability Manager

In order to enable loading and unloading of optional features, we developed a component called *variability manager.* It is responsible for adding and removing binary code of optional features and correctly linking the corresponding methods calls. To this end, it provides three methods to handle features, which can load and unload features and return a function pointer to a feature method:

```
void addFeature(const char* feature_name)
void removeFeature(const char* feature_name)
void* getMethod(const char* method_name)
```

When adding a feature with `addFeature`, the variability manager stores the addresses of all loaded feature methods inside a hash map. With `getMethod`, the application can request a function pointer to a feature method from the variability manager. If a requested method is not available inside the enclave, the variability manager automatically adapts the configuration and loads the corresponding feature and then returns the new function pointer.

Furthermore, the variability manager holds the current configuration of the application and ensures that it is valid at all times. Whenever a feature is added or removed the variability manager updates the configuration and resolves potential conflicts by applying decision propagation. This may result in loading or unloading of other features as well.

The variability manager belongs to the trusted part of the application and is considered a static feature (i.e., it cannot be unloaded). It is initialized together with the enclave and starts with a minimal configuration that only contains the core features.

3.3.1 Loading Features. To load a feature the corresponding feature file is read from memory and parsed to extract the list of features and the binary code. The binary code is copied to a suitable memory address inside the executable heap. The memory address for each method is then stored inside a hash map using the method name as a key. For each method, we compute its memory address by adding its offset value from the feature file to the memory address of the stored binary code.

Whenever a feature method is requested, the variability manager checks whether it is already available within the enclave or whether the corresponding feature must be loaded first. For this purpose, the variability manager stores a list of all features in the feature model, their contained feature methods, and a flag indicating whether a feature is loaded.

3.3.2 Unloading Features. To unload a feature, all its methods are removed from the hash map by simply removing the respective entries. Then the block of memory that were allocated for the feature's binary code is freed.

Unloading a feature can either be done manually by calling `removeFeature` or automatically, if the variability manager resolves a conflict or tries to load a new feature, while there is no memory space left. Then at least one other feature must be removed before loading the new feature.

3.3.3 Replacement Strategies. In order to automatically remove features from the enclave, the variability manager must use a replacement strategy to determine which feature can be removed. There are many different replacement strategies that could be used for this purpose. We require a strategy ensuring that no currently needed feature is removed and enough memory is freed for the feature about to be loaded.

At present, we use the simple yet effective strategy Least-Recently-Used, which starts by removing the feature that was least recently accessed. Features are repeatedly removed until enough space is available to load a new feature and no conflicts remain in the current configuration.

4 IMPLEMENTATION DETAILS

We implemented a rudimentary prototype of our approach in order to evaluate the performance of loading features and executing feature methods. In this section, we describe the implementation details of our prototype and its software dependencies.

4.1 SGX Hardware and Software

Our evaluation system employs an Intel Xeon CPU E3-1230 v5, which features the SGX instruction set. In order to enable SGX on our evaluation system we use the open source Linux* Intel(R) SGX software stack[1] in Version 1.9, which includes the SGX driver, the SGX Software Development Kit (SDK), and the SGX Platform Software (PSW).

However, theres is an issue with our hardware, that would normally prevent our approach from working. Normally, the heap inside an enclave is not executable, as it would create a potential attack surface. This means, the system prevents us from dynamically loading a feature into the heap and then executing a feature method. Most recent Intel CPUs with SGX support allow the software to modify page attributes inside the enclave, which would allow us to make the pages, storing the feature methods executable [20]. However, as we run on slightly older hardware, we need to make a small change to the software stack instead. In particular, we use a patch from Silva et al. [24], which adds an option to make the allocated memory within the heap of the enclave executable. As explained above this increases the attack surface of our approach and, thus, would not be a valid option for a product application. However, in later prototypes of our approach, we can rely on the newer hardware features, to securely acquire executable memory.

[1]https://github.com/intel/linux-sgx

4.2 Loading and Unloading Features

To load a feature, the variability manager must first find the corresponding feature file. As the name schema of the files are always `<feature_name>.f`, they can easily be located within the file system. Second, the file's content must be read and parsed. For file handling we use the SGX SDK versions of the standard file handling methods of C, namely, `sgx_fopen`, which opens a file from memory, `sgx_fread`, which reads a given number of bytes from a file, and `sgx_fclose`, which closes the file. Using these methods also guarantees that the feature files are securely stored as `sgx_fopen` already encrypts/decrypts the specified file. Third, the feature methods must be placed into the map. We use a hash map implementation with linked buckets and the *sha256* algorithm for computing hash sums. For unloading a feature, we simply remove the corresponding entries from the map and free the allocated memory block.

4.3 Current Limitations

At present, there are a number of limitations of our approach that arise either from our utilized hardware, the rudimentary implementation of our prototype, or by our approach itself. In future implementations, we plan to fix most of these limitations to further support developers.

Memory Layout Restrictions. One limitation of our current approach is that methods of one feature must not be fragmented throughout the memory, but need to stay as one coherent block to keep their relative positions. As methods from one feature may call each other directly, their binary code contains relative addresses pointing the corresponding method. Therefore, altering the relative positions can result in undefined behavior. In turn, this may lead to a sub-optimal memory space utilization. Furthermore, our prototype does not yet handle global variables within an application. However, this is only a limitation in our current implementation, as our conceptual approach can easily be extended to allow annotations for global variables as well.

Concurrent Operations. At the moment, our prototype is not thread-safe. This means only one client may run an application at a time. Otherwise race conditions might occur, where one client tries to access a method that was just removed by another client leading to an undefined behavior at runtime. However, this is not an issue of our concept, but does only concern our current prototype.

Limitations on Variability. Our current variability mechanism is limited to method level granularity. Though this is not a severe complication, it may impose unnecessary restrictions to developers. Nevertheless, an apparent finer granularity can be achieved by explicitly including hook methods at suitable positions within a method. Furthermore, our variability mechanism does not yet allow refinement or overwriting of methods.

Performance Degradation. By design, our approach introduces an overhead to calling feature methods, which includes loading features from the unprotected memory, computing positions in the hash map, and indirectly calling methods. This may result in a performance degradation compared to running an application outside an enclave, but also compared to running an application inside a static enclave. To estimated the potential performance impact of our approach, we conduct an evaluation using our implemented prototype.

5 EVALUATION

As our current approach introduces another layer of indirection to many method calls inside the enclave, we are interested in the impact to the overall execution time. To this end, we use a small example application to measure the execution time with and without applying our approach in order to determine the introduced overhead and show the potential benefits and bottlenecks of our approach.

5.1 Setup

We use our running example (cf. Listing 1) as a case study in the evaluation scenario. In particular, we measure the execution time of the method add in different settings.

Evaluation Settings. The method add takes two integers and returns the sum of both. As this process only consists of a few CPU instructions and is rather fast on modern systems, we run a loop that execute this method multiple times. We start by using 1 iteration and do further measurements with 100, 10,000, and 1,000,000 iterations. In each iteration, we call add to add the counting variable i to a given variable a, which we initially set to one:

```
int a = 1;
for (int i = 0; i < number_of_iterations; i++)
    a = add(a,i);
```

In total, we test five different settings:

1. UD **U**ntrusted application, **D**irect call
2. TD **T**rusted application, **D**irect call
3. TI **T**rusted application, **I**ndirect call
4. THI **T**rusted application, **H**ashed **I**ndirect call
5. TLI **T**rusted application, **L**oaded **I**ndirect call

We start with the UD setting, in which we simply execute add directly in a regular C application outside an enclave. Then, with each setting, we introduce a some additional overhead that is necessary for our approach. Starting with TD, we execute add inside an enclave. However, in TD add is just a regular method inside an enclave and not yet a dynamically loaded feature method. With TI, we indirectly execute add by calling a function pointer that points to its binary code. These first three settings, UD, TD, and TI act as a baseline to differentiate the overhead introduced by different mechanics (i.e., ecalls, function pointers, hashing). But they do not fully meet our requirements of securely and dynamically executing the add method. While UD is not secure at all, TD and TI lack the flexibility to execute any method, we load inside the enclave. In the THI setting, we read the function pointer for add from a hash map before executing the method. The TLI setting is similar to THI, but loads the feature *add* before executing add for the first time. These two setting finally behave as we desire. Using these five settings we can compare the impact our approach has on the execution time of a simple method.

Table 3: Median of measured times for each setting.

Iterations	UD	TD	TI	THI	TLI
1	21	4,617	4,618	5,032	4,448,943
100	1,220	21,185	24,864	132,372	4,514,485

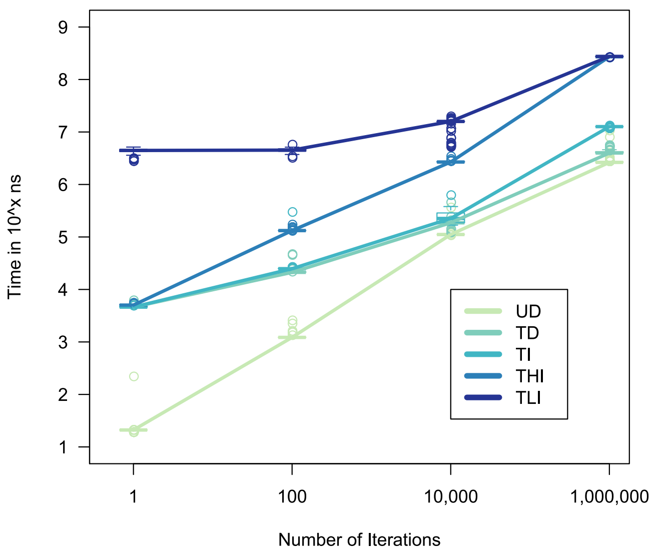

Figure 3: Median of execution times for function add in different settings for different number of iterations.

Further Details. We run our evaluation on a system with the following specifications:

- **CPU**: Intel(R) Xeon(R) CPU E3-1230 v5 @ 3.40GHz
- **Memory**: 32 GB
- **OS**: Ubuntu 16.04

For the evaluation, we disabled all compiler optimizations (i.e.,-O0), as otherwise the complier may modify the evaluation code (e.g., by loop unrolling), thereby biasing the results. In order to further mitigate potential measurement biases, we execute each setting 100 times to calculate an average value. Each time we measure the execution time for one complete ecall. Except for UD, where we simply measure the execution of add.

5.2 Results

In Table 3, we present our measured execution times (in nano seconds) of each setting with 1 and 100 iterations of add. Each data point is the median of 100 measurements. Additionally, we depict the result for more iterations of add in Figure 3. In this figure, we also include all measured values in form of box plots, with lines that connect the corresponding medians. Note that, both axis of the diagram have a logarithmic scale.

From the data, we can clearly see that, indeed, the execution time increases with each new setting due to the additional overhead introduced. As expected, UD has a very short execution time (21 ns), which is two orders of magnitude faster than TD (4,617 ns). We can explain this slowdown with the overhead of the ecall and the general overhead of running code inside an enclave. Surprisingly, introducing an indirect method call via a function pointer does not add much overhead, as we can see by comparing TD (4,617 ns) and TI (4,618 ns). The slowdown is only noticeable when executing many iterations of add (e.g., 21,185 ns vs. 24,864 ns for 100 iterations). Regarding THI, we can see that the lookup inside a hash map has a higher impact on the performance (5,032 ns). That is the case, because it takes some time to compute the hash sum and search for the correct entry in the map. Loading a feature from memory has by far the highest impact on performance, as we can see for TLI (4,448,943 ns). However, in case the feature is not removed from the enclave, this slowdown only occurs once per feature. For subsequent calls to the feature method add the performance is the same as for FHI. We can observe this in Figure 3, as the execution times of THI and TLI get closer the more iterations are executed.

Overall, the evaluation results show much potential for our approach. The main bottleneck for performance is loading a feature into the enclave. Thus, if we are able to reduce loading processes to a minimum the benefit of having more space available in the EPC and, thus, doing fewer paging can outweigh the loading cost during runtime. Of course, this depends highly on the use case scenario. An application with a relatively stable configuration should perform much better than an application that executes many different tasks simultaneously.

5.3 Threats to Validity

There are some issues that could threat the validity of our evaluation results. Firstly, we used a rather small, non-real-world application. This may lead to a result that is not generalizable to other cases. While this is true, we did not aim for a general statement about the performance of our approach, but mainly wanted to show its potential benefits and possible downsides.

Secondly, our tested method has only a short execution time, the measurement could be imprecise and depends heavily on the resolution of the evaluation system's internal clock. We performed 100 measurements for each setting in order to mitigate possible measurement errors.

Finally, there is always the issue of potential programming mistakes within the implementation. This could lead to invalid measurements and bias the overall result. Although we cannot guarantee that our implementation is bug free, due to the simplicity of the application, we are confident that it functioned correctly. In addition, we examined the individual results of the add method to ensure that it computed the correct value.

6 RELATED WORK

Silva et al. [24] implemented a similar concept for dynamically loading a user-defined method into an enclave and executing it. Indeed, we use a very similar mechanism to dynamically execute a method. However, their approach is not based on the notions of features, which can be loaded simultaneously and influence each other and also lacks the dynamic self-adaptation that we achieve with our approach.

There are other concepts for running entire applications inside an enclave, such as Haven [7], SCONE [5], and Graphene [26]. These approaches focus on running unmodified applications with SGX. This is similar to our approach, as we also aim to include a complete application in the enclave. However they do not provide a mechanism to update the enclave during runtime.

Another related topic is the automated partitioning of applications with Glamdring by Lind et al. [19]. Their goal is to keep the size of the trusted code base, which runs inside an enclave as minimal as possible. While we use the opposite approach (i.e., executing the entire application in the enclave), such a partitioning algorithm can be employed by developers to identify useful features for the decomposition of an application into a product line.

There are several other dynamic variability mechanisms in the context of DSPLs such as using services [1, 18], components [17], and dynamic composition [23]. While these can be applied well for many applications, we must take care of the restrictions imposed by using SGX, for instance being limited to the heap memory for loading features and the need to encrypt and verify features.

7 CONCLUSION AND FUTURE WORK

In this paper, we present an approach for the dynamic adaption of binary code inside an Intel SGX enclave based on dynamic software product lines. Our approach allows to freely load and remove features into and from an enclave at runtime. This enables an application to remove unnecessary code from the limited protected memory and, thereby, including more data for processing, with the ultimate goal of increasing performance. We implemented a prototype of our approach and evaluated its performance impact on loading features and executing feature methods inside an enclave. Our evaluation results show that, though loading a feature is rather costly, the execution of feature methods is almost neglectable compared to executing a regular method inside an enclave. Thus, we are looking forward to further extend our approach and prototype.

In future work, we intend to address the current limitations of our approach. Once we overcame these limitations and built a fully-fledged prototype, we want to evaluate it using real-world systems. In particular, we plan to test out our approach using database systems, as they required lots of data inside the EPC for data processing and, thus, we hope to increase their performance by increasing the available EPC space. Moreover, we want to increase the performance of the resulting dynamic product line by improving our variability mechanism. For instance, a promising approach is to directly patch jump addresses in the binary code, instead of relying on indirect method calls via a hash map implementation. Another important issue that we want to investigate is assisting the user in decomposing the application into meaningful features for usage inside an enclave. This is especially interesting, because meaningful decomposition might be different from a developer point of view than from the aspect of runtime performance.

ACKNOWLEDGMENTS

We thank Nico Weichbrodt and Rüdiger Kapitza for their valuable help. This research is supported by DFG grants LE 3382/3-1 and KA 3171/6-1.

REFERENCES

[1] Germán H. Alférez, Vicente Pelechano, Raúl Mazo, Camille Salinesi, and Daniel Diaz. 2014. Dynamic adaptation of service compositions with variability models. *Journal of Systems and Software* 91 (2014), 24–47.

[2] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *International Workshop on Hardware and Architectural Support for Security and Privacy*, Vol. 13. ACM.

[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.*

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A View of Cloud Computing. *Communications of the ACM* 53, 4 (2010), 50–58.

[5] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *USENIX Conference on Operating Systems Design and Implementation.* USENIX Association, 689–703.

[6] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *International Software Product Line Conference.* Springer, 7–20.

[7] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM Transactions on Computer Systems* 33, 3 (2015), 8:1–8:26.

[8] Nathalie Brender and Iliya Markov. 2013. Risk Perception and Risk Management in Cloud Computing: Results from a Case Study of Swiss Companies. *International Journal of Information Management* 33, 5 (2013), 726–733.

[9] TIS Committee et al. 1995. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2. *TIS Committee* (1995).

[10] Victor Costan and Srinivas Devadas. 2016. *Intel SGX Explained.* Technical Report. IACR Cryptology ePrint Archive.

[11] Krzysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications.*

[12] Krzysztof Czarnecki and Andrzej Wąsowski. 2007. Feature Diagrams and Logics: There and Back Again. In *International Software Product Line Conference.* IEEE, 23–34.

[13] Prashant Gupta, Arumugam Seetharaman, and John Rudolph Raj. 2013. The usage and adoption of cloud computing by small and medium businesses. *International Journal of Information Management* 33, 5 (2013), 861–874.

[14] Mike Hinchey, Sooyong Park, and Klaus Schmid. 2012. Building Dynamic Software Product Lines. *Computer* 45, 10 (2012), 22–26.

[15] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. 2013. Using innovative instructions to create trustworthy software solutions. *International Workshop on Hardware and Architectural Support for Security and Privacy* 11 (2013).

[16] Sebastian Krieter, Jacob Krüger, Nico Weichbrodt, Vasily A. Sartakov, Rüdiger Kapitza, and Thomas Leich. 2018. Towards Secure Dynamic Product Lines in the Cloud. In *International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER '18).* ACM, New York, NY, USA, 5–8.

[17] Jaejoon Lee and K. C. Kang. 2006. A Feature-oriented Approach To Developing Dynamically Reconfigurable Products In Product Line Engineering. In *International Software Product Line Conference.* 131–140.

[18] Jaejoon Lee, Dirk Muthig, and Matthias Naab. 2008. An Approach for Developing Service Oriented Product Lines. In *International Software Product Line Conference.* 275–284.

[19] 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. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *USENIX Annual Technical Conference.* USENIX Association, 285–298.

[20] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. 2016. Intel&Reg; Software Guard Extensions (Intel&Reg; SGX) Support for Dynamic Memory Management Inside an Enclave. In *International Workshop on Hardware and Architectural Support for Security and Privacy.* ACM, 10:1–10:9.

[21] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. *International Workshop on Hardware and Architectural Support for Security and Privacy* 10 (2013).

[22] Marko Rosenmüller, Norbert Siegmund, Mario Pukall, and Sven Apel. 2011. Tailoring Dynamic Software Product Lines. In *International Conference on Generative Programming and Component Engineering.* ACM, 3–12.

[23] Marko Rosenmüller, Norbert Siegmund, Gunter Saake, and Sven Apel. 2008. Code Generation To Support Static And Dynamic Composition Of Software Product Lines. In *International Conference on Generative Programming and Component Engineering (GPCE '08).* ACM, New York, NY, USA, 3–12.

[24] Rodolfo Silva, Pedro Barbosa, and Andrey Brito. 2017. DynSGX: A Privacy Preserving Toolset for Dynamically Loading Functions into Intel(R) SGX Enclaves. *ArXiv e-prints* (2017).

[25] Subashini Subashini and Veeraruna Kavitha. 2011. A Survey on Security Issues in Service Delivery Models of Cloud Computing. *Journal of Network and Computer Applications* 34, 1 (2011), 1–11.

[26] Chia-Che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX Annual Technical Conference.* USENIX Association, 645–658.