Privagic: automatic code partitioning with explicit secure typing

Subashiny Tanigassalame, Yohan Pipereau, Adam Chader, Jana Toljaga, Gaël Thomas

Privagic: automatic code partitioning with explicit secure typing

Subashiny Tanigassalame
subashiny.tanigassalame@inria.fr
Inria Saclay
Palaiseau, France

Yohan Pipereau
yohan.pipereau@telecom-sudparis.eu
Telecom SudParis, IP Paris
Palaiseau, France

Adam Chader
adam.chader@telecom-sudparis.eu
Telecom SudParis, IP Paris
Palaiseau, France

Jana Toljaga
jana.toljaga@telecom-sudparis.eu
Telecom SudParis, IP Paris
Palaiseau, France

Gaël Thomas
gael.thomas@inria.fr
Inria Saclay
Palaiseau, France

ABSTRACT

Partitioning a multi-threaded application between a secure and a non-secure memory zone remains a challenge. The current tools rely on data flow analysis techniques, which are unable to handle multi-threaded C or C++ applications. To avoid this limitation, we propose to trade the ease-of-use of data flow analysis for another language construct: explicit secure typing. With secure typing, as with data flow analysis, the developer annotates memory locations that contain sensitive values. However, instead of analyzing how the sensitive values flow, we propose to use these annotations to only check typing rules, such as ensuring that the code never stores a sensitive value in an unsafe memory location. By avoiding data flow analysis, the developer has to annotate more memory locations, but the partitioning tool can handle multi-threaded C and C++ applications.

We implemented our explicit secure typing principle in a compiler named Privagic. Privagic takes a legacy application enriched with secure types as input. It outputs an application partitioned for Intel SGX. Our evaluation with micro- and macro-applications shows that (i) explicit secure typing can handle multi-threaded C and C++ applications, (ii) adding explicit secure types requires a modest engineering effort of less than 10 modified lines of codes in our use cases, (iii) using explicit secure typing is more efficient than embedding a complete application in an enclave both in terms of performance and security in our use cases.

CCS CONCEPTS

• **Security and privacy → Trusted computing**; • **Software and its engineering → Compilers**.

KEYWORDS

Trusted computing, secure typing, code partitioning

ACM Reference Format:
Subashiny Tanigassalame, Yohan Pipereau, Adam Chader, Jana Toljaga, and Gaël Thomas. 2024. Privagic: automatic code partitioning with explicit secure typing. In *25th International Middleware Conference (MIDDLEWARE '24), December 2–6, 2024, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3652892.3700759

MIDDLEWARE '24, December 2–6, 2024, Hong Kong, Hong Kong

ACM ISBN 979-8-4007-0623-3/24/12
https://doi.org/10.1145/3652892.3700759

1 INTRODUCTION

Confidential computing consists of protecting user data when it is processed in an untrusted system such as a cloud infrastructure [3, 13, 14, 29]. At the hardware level, confidential computing relies on a *trusted execution environment* (TEE) (Intel's SGX [16], AMD's SEV [22] or ARM's TrustZone [2, 31]). A TEE is a hardware environment that isolates a memory zone, named an *enclave*, from a potentially compromised operating system or hypervisor. For that, a TEE relies on remote attestation for authentication, and on hardware cryptography to enforce the integrity and the confidentiality of both the code and data contained in an enclave.

Since manually partitioning an application between an enclave and the unsafe memory is difficult, many automatic partitioning tools have been proposed [9, 20, 23–26, 34, 42, 45–47]. With these tools, the developer annotates some sensitive values, and the tool then analyses the code to find which memory locations the sensitive values flow into. Based on this analysis, the tools then partition the data and the code of the application between the trusted and the untrusted parts. For a modest engineering effort, these tools avoid thus the large attack surface of the frameworks that load a whole application inside an enclave [5, 8, 27, 32, 41] (see §9.2.2).

While the tools based on data flow analysis are promising, they fail to handle a multi-threaded application written in the C or C++ language. Section §3 explains in detail why, but, at a high level, the main issue is that data flow analysis analyses an application *sequentially*. As a result, a data flow analysis tool does not see the pointer modifications executed in parallel by the other threads. The tool can wrongly conclude that a pointer points to an enclave because this observation is correct if the code executes sequentially, but not if the code executes in parallel. With multiple threads, data flow analysis can thus let a sensitive value escape in unsafe memory through a pointer wrongly identified as pointing to an enclave.

Since data flow analysis fails to handle a multi-threaded C or C++ application, we propose to start from another point in the design space. We propose to let the developer explicitly annotate all the memory locations that contain sensitive values. Since the developer explicitly annotates all the sensitive memory locations, there is no need for code analysis. We avoid thus by construction any risk of analysis errors in a multi-threaded application. However, by trading

```
1.  struct account {
2.    char color(blue) name[256];
3.    double color(red) balance;
4.  };

5.  struct account* create(char* name) {
6.    struct account* res = malloc(sizeof(*res));
7.    strncpy(res->name, name, 256);
8.    res->balance = 0.0;
9.    return res;
10. }
```

Figure 1: A simple example of the Privagic language in C.

automatic analysis for manual annotation, we also trade the ease of use of data flow analysis for extra work for the developer. The evaluation presented in this paper aims to verify that the burden placed on the developer results in a reasonable engineering effort.

To allow the developer to indicate the memory locations that contain sensitive values, we introduce a new language construct named a *secure type*. A secure type is a type enriched with an enclave identifier, which we name *color*. Figure 1 illustrates the principle with the C language. Starting from a legacy application, the developer adds an identifier to the types of the fields `name` (line 2) and `balance` (line 3). These identifiers mean that the `name` field lives in an enclave named blue and that the `balance` field lives in an enclave named red.[1]

By explicitly adding a secure type to each sensitive memory location, partitioning the code becomes straightforward. For example, in Figure 1, since line 7 manipulates a blue field, the line has to be executed in the blue enclave. Similarly, line 8 has to be executed in the red enclave. Lines 6 and 9 do not access any color. They can thus safely be executed in any enclave or outside any enclave.

Secure typing indicates how to partition the code. However, by itself, secure typing does not provide any security guarantee. For example, exactly as an integer value can be stored in a float variable, a sensitive colored variable could be stored in unsafe memory (i.e., a memory location with the color of unsafe memory). To enforce security, we propose thus complement secure typing with *typing rules*. These rules first ensure confidentiality by ensuring that a sensitive value cannot escape from its enclave. At a high level, the rules report an error if an instruction stores a colored value in a memory location with a different color. For integrity, the rules ensure that a store to a colored memory location is generated in the enclave of the memory location. The rules also prevent Iago attacks [11], which consist of sending a poisoned value to an enclave. For that, the rules report an error if an instruction takes inputs with two different colors.

Explicitly typing each memory location that may contain a sensitive value makes the partitioning of a multi-threaded application possible. However, adding a secure type to each sensitive memory location can be time-consuming for the developer. For this reason, we propose to ease the use of secure typing with a simple form of type inference. In detail, we propose to infer the type of an uncolored local variable, but only if the code does not create a pointer to the variable. In such a case, the variable does not escape from the scope of a single function, which avoids inter-procedural analysis. Moreover, since the variable does not escape from the scope of its function, the variable cannot be accessed by another thread. With this restriction, inferring a secure type requires a simple use-def chain analysis, and the deduced type is correct even in multi-threaded applications.

We implemented our principle of secure typing in the Privagic compiler for Intel SGX and the C/C++ languages. The Privagic compiler relies on the LLVM compiler, which means that it does not rely on the C/C++ semantic: it considers a low-level intermediate representation of the code with secure types added to the variables, the arguments, and the fields of the data structures. To verify that using Privagic requires a decent engineering effort, we evaluated Privagic with a complete legacy application (memcached) and several common data structures. Overall, our evaluation shows that:

- Secure typing correctly partitions a multi-threaded C and C++ application, while this is not the case with data flow analysis.
- Protecting common data structures requires the modification of 6 lines of code or less. Protecting the central data structure of memcached requires the modification of 9 lines of code. With data flow analysis, the authors of Glamdring [23] report a modification of 2 lines of code to protect the same data structure. Secure typing is thus more intrusive than data flow analysis, but the engineering effort remains decent.
- Partitioning significantly improves performance and reduces the trusted computing base compared to fully embedding a complete application in an enclave, as done, for example, in Scone [5]. In particular, Privagic is up to 10 times faster than Scone, and it reduces the trusted computing base within each partition by a factor of more than 200. This observation is consistent with performance assessments conducted by previous work on other partitioning tools, such as Glamdring [23].
- Thanks to its accuracy, explicit secure typing can be used to partition an application in more than one partition by annotating variables with different colors. We explored this possibility, but we had a negative result. In detail, Privagic can generate applications with multiple colors, but the generated code is not protected against Iago attack. Strongly protecting multi-color applications is probably possible, but would require the use of authenticated pointers, which we let as future work.

To summarize, the paper makes the following contributions: (i) it introduces explicit secure typing, which makes the automatic partitioning of multi-threaded C or C++ applications possible, (ii) it proposes the Privagic compiler, which relies on explicit secure typing to automatically partition an application for Intel SGX, and (iii) it shows that Privagic is easy to use, is efficient, and reduces the trusted computing base.

The rest of the paper is organized as follows: §2 presents the background, §3 motivates our work by analyzing the related work, §4 presents our threat model, §5 gives an overview of Privagic, §6 presents our secure type system, §7 shows how Privagic partitions the code, §8 discusses the limitation, §9 presents the evaluation, and §10 concludes.

[1] Note that we name the identifier a color to simplify the presentation in the paper, but the user can use any identifier.

```
1.  @y = global i32 0           ; int y = 0;

2.  define i32 @test(i32 %0)  { ; int test(int a) {
3.    %2 = alloca i32           ;   int x;
4.    %3 = add i32 %0, 42       ;
5.    store i32 %3, i32* %2     ;   x  =   a + 42;
6.    store i32 %3, i32* @y     ;   y  =   a + 42;
7.    %4 = call i32 @f(i32* %2) ;
8.    ret i32 %4                ;   return f(&x);
9.  }                           ; }

10. declare i32 @f(i32*) #1     ; extern int f(int*);
```

Figure 2: Registers and memory in LLVM.

```
1.  int a;
2.  int b;
3.  int* x;

4.  // s is sensitive
5.  void f(int s) {
6.    x = &a;
7.    *x = s;
8.  }

9.  void g() {
10.   x = &b;
11. }
```

(a) Data flow analysis

```
11. int color(blue) a;
12. int b;
13. int color(blue)* x;

14. // s is blue
15. void f(int s) {
16.   x = &a;
17.   *x = s;
18. }

19. void g() {
20.   x = &b; // FAIL
21. }
```

(b) Explicit secure typing

Figure 3: Hidden pointer modification (f and g runs in parallel).

2 BACKGROUND

This section presents the background: Intel SGX and LLVM.

2.1 Intel SGX

Intel SGX protects memory zones named enclaves by defining two processor modes: an enclave mode and a normal mode. In normal mode, the processor prevents access to the memory of the enclaves. When the processor enters the enclave mode, it gains access to a single enclave, which we name the active enclave. In enclave mode, the processor can access the memory of the active enclave and the memory located outside any enclave, which we name the unsafe memory. The processor can, however, not access the memory of the non-active enclaves in enclave mode.

To protect an enclave, Intel SGX leverages read/write page permissions. Intel SGX also ensures that a device under the control of an attacker cannot read the memory of an enclave in DMA (direct memory access). For that, Intel SGX encrypts a cache line when it leaves the CPU package. In its first version, Intel SGX also enforces the integrity of the enclaves despite a device under the control of an attacker. For that, Intel SGX maintains a tree of cryptographic hashes, which is used to detect a write to an enclave in DMA [39]. In its second version, Intel SGX does not enforce integrity if a device is under the control of an attacker.

2.2 LLVM

LLVM considers an abstract machine with a memory and an infinite number of typed registers. An LLVM instruction takes registers as input and outputs a new register, which means that a register is assigned once (i.e., Single Static Assignment representation [1]). As a result, an instruction and its output register are equivalent.

Figure 2 presents an example of LLVM IR. A typical instruction is the `add` instruction at line 4, which adds 42 to the register `%0` (the parameter of the function). This instruction outputs its result in the register `%3`.

Additionally to the registers, a code can access memory with the `load` and `store` instructions (e.g., lines 5 and 6 in Figure 2). The code can create a memory location in the heap with `malloc`, a local variable with `alloca` (line 3), and a global variable with `global` (line 1). In each case, the instruction returns a register that points to the variable.

As illustrated in Figure 1 (see §1), with Privagic, the developer can enrich a type with a color. The `color` keyword is a macro. It transforms the color declaration into a generic C annotation.[2] The `clang` frontend does not handle itself the annotation: it simply emits the annotation in the LLVM IR. Privagic uses these annotations to identify the colors associated to the types.

3 MOTIVATION AND RELATED WORKS

In order to ease the use of SGX, several frameworks propose to run a complete application with its dependencies in an enclave [5, 8, 27, 32, 41]. These frameworks ease the use of SGX, but they lead to a large TCB that can reach tens of megabytes (see §9.2.2).

To avoid a large TCB, two techniques are proposed. The first technique consists of defining new language abstractions or new programming models [36, 37]. These abstractions ensure a high level of safety, but they require a complete rewriting of the application, which makes them inadequate for legacy applications with a large code base. Similar language abstractions were proposed to ensure the confidentiality of sensitive values in the context of a distributed system [12, 49, 50]. They also require a complete rewriting of the application, which also makes them inadequate for legacy applications.

The second technique consists of automatically partitioning an application. Privagic belongs to this category. Table 1 gives an overview of the automatic partitioning techniques used in previous works. These works target Intel SGX, AMD's SEV, and ARM TrustZone, but also privilege separation [35], which consists of isolating in a second process the most dangerous functions of an application. All these works are based on data flow analysis techniques. The developer annotates sensitive variables or functions, and then, a tool analyzes how the data flows in the application: by using use-def chains [1], abstract interpretation [17], in-vitro execution [45, 46], points-to analysis [4, 7, 18, 33, 38, 44], program dependence graphs [19] or taint analysis [6, 25].

As illustrated in Figure 3.a, data flow analysis cannot handle a multi-threaded application in C or C++. In this example, we suppose that `s` is a sensitive value. By analyzing lines 6 and 7, a data flow analysis tool easily deduces that `a` contains a sensitive value. However, since a data flow analysis tool analyzes the code sequentially, the tool does not see that line 10, which may be executed in parallel by another thread, can change the pointer `x` from `a` to `b` if line 10 is executed between lines 6 and 7. Since exploring all the possible thread inter-leavings is not possible because of the combinatorial

[2] E.g., `__attribute__((annotate ("blue")))`.

Tool	Technique	Language	Starting point	Partitioning granularity		Multiple threads	Language coverage
				Code	Data		
Glamdring [23]	Abstract interpretation [17]	C	Func. args.	Function	Global variable	No	Complete
Privtrans [9]	Use-def chains [1]	C	Function	Function	Incorrect[1]	No	Incomplete[1,4]
Treillis [26]	Call graph [1]	C	Func./glob. var.	Function	Incorrect[1]	No	Incomplete[1]
ProgramCutter [45]	In-vitro execution	C	None	Function	Incorrect[1]	No	Incomplete[1,4]
SeCage [25]	Taint analysis	C	Local variable	Function	Incorrect[1]	Yes	Incomplete[1]
Montsalvat [47]	Points-to analysis [4]	Java	Function	Java class	Java class	Yes	Complete
Civet [42]	Points-to analysis [4]	Java	Java class	Java class	Java class	Yes	Complete
Rubinov at al. [34]	Taint analysis [6]	Java	Variable	Java class	Java class	No	Incomplete[3]
GoTEE [20]	Prog. dependence graph [19]	Go	goroutine	goroutine	Global variable	Yes	Complete
PtrSplit [24]	Prog. dependence graph [19]	Agnostic	Global variable	Function	Global variable	No	Complete
SecV [46]	In-vitro execution	Agnostic	Function	Function	Object	No	Incomplete[4]
Our contribution	Language typing	C/C++	Type	Instruction	Field	Yes	Complete

[1]: does not support pointers
[2]: does not support thread local storage
[3]: the paper states that the technique can only handle 86% of the applications
[4]: does not inspect all the code

Table 1: Automatic partitioning tools. No tool can handle multi-threaded applications in the general case.

explosion, the previous tools misbehave for a multi-threaded application.

This limitation is clearly identified in previous work (see column "Language coverage" of Table 1): page 52 of [10] for the abstract interpretation engine used in Glamdring [23], [6] for the taint analysis used in Rubinov et al., and [19] for the program dependence graph used in PtrSplit [24].

Despite the limitation of data flow analysis with multiple threads, four tools can, however, handle multi-threaded applications: SeCage [25], Montsalvat [47], Civet [42] and GoTEE [20]. However, they cannot handle multi-threaded C and C++ applications in the general case.

SeCage can handle multi-threaded C applications because it does not handle pointers at all. As a result, SeCage cannot be used with most of the C applications.

Montsalvat and Civet can handle multi-threaded applications by relying on a sort of language typing simpler than the language typing proposed in Privagic. Technically, with Montsalvat and Civet, the developer annotates the Java classes that have to be stored in an enclave. These annotations are enough for the Java language because Montsalavat and Civet consider that all the fields of an object are private, and because a Java application cannot create a direct pointer to one of the fields inside an object. In C and C++, the code can create such a pointer. These pointers lead to a much more complex interaction between the code and the sensitive value: any pointer may potentially access a sensitive value, which is not a case handled by Civet or Montsalvat. For this reason, Privagic goes one step further by introducing the concept of secure typing, which makes pointer analysis doable.

GoTEE can handle threads because it considers goroutines that are not supposed to share memory. This hypothesis does not hold for a language such as C or C++, in which memory sharing between the threads is allowed.

Privagic avoids the limitation of data flow analysis and can handle multi-threaded C and C++ applications in the general case. With the same example (see Figure 3.b), if the developer forgets to explicitly color b in blue, the compiler will report a type error at line 20. In detail, at line 20, x is a pointer to a blue value, but &b is a pointer to an uncolored memory location. Privagic detects thus a type error because storing a pointer to an uncolored memory location in a pointer to a colored memory location is prohibited, exactly as storing a pointer to a float in a pointer to an integer is prohibited.

```
1. int x = 0, y = 0;
2. int color(blue) b;

3. void f() {
4.   if(b == 42) // basic block A
5.     x = 1;    // basic block B (indirect leak)
6.   y = 2;      // basic block C (non sensitive)
7. }
```

Figure 4: Indirect color propagation.

4 SECURITY GUARANTEES

We suppose an attacker that fully controls a machine (operating system, hypervisor, and hardware included). We suppose that the attacker cannot read or write the memory of the enclaves protected by Intel SGX. For that, we suppose that the processor, the Privagic runtime, and the software development kit provided by Intel to use SGX are correct and do not contain bugs. We do not consider side-channel attacks since Intel SGX does not address this attack vector.

With such an attacker, Privagic aims to enforce confidentiality and integrity, as well as to prevent Iago attacks. The attacker should not be able to (i) directly or indirectly extract a sensitive value from an enclave, (ii) modify a sensitive value, or (iii) alter the normal behavior of an enclave by passing it a corrupted value.

Enforcing confidentiality. Privagic enforces confidentiality by starting from a fundamental property. **A memory location or an LLVM register can have at most one color**. This property ensures that a sensitive value can only belong to a single enclave. To enforce this property, Privagic prevents the use of a union with fields with different colors, as this would amount to a sensitive value with two colors.

Starting from this property, Privagic ensures that an attacker that controls the operating system cannot deduce the color of a

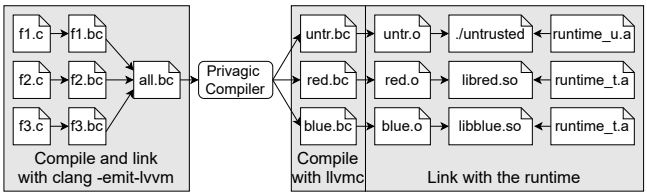

Figure 5: Overview of Privagic.

sensitive value with five rules. The two first rules ensure that a colored value cannot directly escape from its enclave, while the three last rules ensure that a colored value cannot indirectly escape from its enclave.

In detail, the first rule ensures that a colored value cannot escape from its enclave through a store. For that, Privagic ensures that storing a colored value in a memory location with a different color (or without a color) is prohibited.

The second rule ensures that if an instruction consumes or outputs a value with the color C, then the instruction is executed in C. This rule ensures that only the enclave C can access a value with the color C.

The third rule prevents an explicit indirect leak. It ensures that if an instruction takes a colored value as input, then the output has the same color. With this rule, an attacker cannot deduce the value of a colored value by observing a value that was computed with it.

The fourth rule ensures that a color is preserved despite a pointer cast. This rule states that if a pointer `p` points to a C memory location, `p` is itself C. The code can thus not assign `p` to a pointer `q` pointing to a memory location with a different color. This prevents any possibility of using a cast to change a color.

The fifth rule prevents an implicit indirect leak through a conditional jump. Figure 4 illustrates the issue. In this code, if an attacker observes that `x` is equal to 1, the attacker can deduce that `b` is equal to 42. In order to avoid this leak, Privagic ensures that `x` has a color compatible with `b` (see the details in §6.1.1).

Enforcing authenticity. To enforce authenticity, Privagic ensures that a code executed outside an enclave cannot modify the memory of the enclave. For that, Privagic ensures that a store to a memory location with the color C is necessarily executed in the enclave C.

Preventing Iago attack. To prevent Iago attacks, Privagic ensures that an instruction executed in an enclave never uses a value computed outside the enclave. For that, Privagic ensures that an instruction executed in an enclave C cannot take an input with a different color or without color.

5 OVERVIEW

The Privagic compiler partitions the code between the enclaves and the unsafe memory. It is not specifically tailored for Intel SGX. It supposes a memory shared between the enclaves and the unsafe code, and a runtime that offers inter-enclave communication primitives (see §7.3).

The Privagic compiler can run in two modes. In hardened mode, it enforces confidentiality and integrity, and prevents Iago attacks. In relaxed mode, it only enforces confidentiality and integrity.

Color	Given to	Compatible with
F (Free)	Registers and instructions	Any other color
U (Unsafe)	Memory locations	No color
S (Shared)	Memory locations	No color (but becomes F when loaded)

Table 2: Initial colors given to the uncolored elements.

As shown in Figure 5, the compiler takes a LLVM bitcode file as input. This LLVM bitcode file contains the whole LLVM Intermediate Representation (IR) of the application. It is generated by a classical toolchain, e.g., `clang` for the C language.

Privagic first analyzes the LLVM IR in order to prevent confidentiality, integrity, or Iago issues. Privagic then partitions the LLVM IR into several bitcode files, which are used to generate an executable with a classical compilation toolchain.

5.1 Type inference

As stated in §1, Privagic eases the use of secure types by using type inference in simple cases. In detail, Privagic infers the color of the LLVM register and of the local variables. For the local variables, Privagic only infers the color if the code does not create a pointer to the local variable. Such registers and variables can only be accessed by a single thread, which ensures that the inferred color is correct in multi-threaded applications.

To simplify type inference, Privagic starts by executing the `mem2reg` pass of LLVM. This pass promotes a local variable into a register, except if the code creates a pointer to the local variable. As a result, after the `mem2reg` pass, Privagic just has to infer the colors of the registers to infer the colors of both the registers and the local variables without a pointer.

5.2 Stabilizing algorithm

During the type analysis phase, since Privagic uses type inference, Privagic may change the color of an input of an already visited instruction. This happens in the case of a backward jump (a loop) or of a recursive call. For this reason, Privagic uses a stabilizing algorithm. In detail, Privagic runs one or several full analysis passes on the whole IR of the application. At the end of a pass, if Privagic inferred new colors during the pass, it restarts a new complete analysis pass. Otherwise, the stabilizing algorithm stops.

5.3 Initial colors

After the `mem2reg` pass, if an LLVM element does not have a color, Privagic considers two cases. For an uncolored register or instruction, as shown in Table 2, the element takes the color F (free). This color indicates that the color will be deduced by type inference. If a register or an instruction is still F at the end of the analysis, it means that the element is not bound to any enclave. Privagic replicates the computation of a F register in each enclave, which ensures that using a F register is safe in any enclave. For that reason, F is the only color compatible with any other color.

For a memory location, if the location has an explicit color, Privagic uses it. Otherwise, Privagic considers that the memory location is located in unsafe memory. The memory location takes the color U (untrusted) in hardened mode and the color S (shared) in

Instruction	Color propagation		#
	Registers	Instruction	
$r = load(p)$	$\overline{*p} \sim \overline{p} \wedge (\overline{*p} \neq \text{S} \Rightarrow r \leftarrow \overline{*p})$	$ins \leftarrow \overline{*p}$	1
$r = op(x_0, \ ...)$	$\forall i, r \leftarrow \overline{x_i}$	$ins \leftarrow \overline{r}$	2
$store(p, r)$	$\overline{*p} \sim \overline{p} \wedge \overline{r} \sim \overline{*p}$	$ins \leftarrow \overline{*p}$	3
$x_n = ins(x_0, \ ...)$	$ins \in \mathcal{B} \Rightarrow x_n \leftarrow \overline{\mathcal{B}}$	$ins \leftarrow \overline{\mathcal{B}}$	4

$\overline{x} \sim \overline{y} \Leftrightarrow \overline{x}$ is compatible with $\overline{y} \Leftrightarrow ((\overline{x} \neq \overline{y}) \wedge (\overline{x} \neq \text{F}) \wedge (\overline{y} \neq \text{F}) \Rightarrow$ error)
$x \leftarrow \overline{y} \Leftrightarrow (\overline{x} \sim \overline{y}) \wedge (\overline{x} = \text{F} \Rightarrow x$ takes the color $\overline{y})$
ins: any instruction *op*: *ins* that is not a store, a load or a call

Table 3: Secure type system. $\overline{x}$ is the color of a register x.[3]

relaxed mode. Both colors are incompatible with any other color. Compared to U, S has a special property: it becomes F when it is loaded from memory into a register. Since a value loaded from U remains U, an instruction that belongs to an enclave C cannot use it, which prevents Iago attack. In relaxed mode, Privagic does not protect against Iago attack: a value loaded from S becomes F, which is compatible with C, and can thus be consumed by an instruction executed in C.

6 SECURE TYPE SYSTEM

The type analysis phase of Privagic has two goals. First, it has the goal of preventing confidentiality, integrity, or Iago issues. Then, it has the goal of assigning a color to each instruction. This color is used in the partitioning phase to partition the application.

6.1 Typing rules

Table 3 presents the rules used by Privagic when we ignore the function calls. The rules ensure security by identifying the compatible registers. We say that the registers x and y are compatible if their colors $\overline{x}$ and $\overline{y}$ are compatible, which we note $\overline{x} \sim \overline{y}$. This is the case if $\overline{x}$ is equal to $\overline{y}$, or if $\overline{x}$ or $\overline{y}$ is equal to F. Otherwise, they are incompatible and Privagic reports an error if they should be.

To simplify the presentation, in the remainder, we consider a color C such that C ≠ F. C can be U or S.

6.1.1 Hardened mode. In hardened mode, the unsafe memory is U, which behaves as any other color. For this reason, we consider the unsafe memory as any other enclave. Since the S color does not exist in hardened mode, the $\overline{*p} \neq$ S in Rule 1 is always true, and we can simplify Rule 1 in $\overline{*p} \sim \overline{p} \wedge \overline{r} \leftarrow \overline{*p}$.

Confidentiality - direct leak and explicit indirect leak. To prevent an explicit indirect leak, Rules 1 to 3 ensure that color is preserved instruction after instruction (Rule 2 with $\overline{r} \leftarrow \overline{x_i}$), from a load (Rule 1 with $\overline{r} \leftarrow \overline{*p}$), up to a store (Rule 3 with $\overline{*p} \sim \overline{r}$).

Rules 1 to 3 also ensure confidentiality by ensuring that only an instruction generated in C can access a C register or a C memory location. For a memory location, a load or store that accesses a C memory location is generated in C (fourth column). For a register, the rules ensure that a C instruction only accesses C registers. This is the case of a load or a store thanks to $\overline{p} \sim \overline{*p}$: if $\overline{p}$ = C, then $\overline{*p}$ = C and the instruction is generated in C (fourth column). This is also the case of an operation (Rule 2) thanks to $\overline{r} \leftarrow \overline{x_i}$: if $\overline{x_i}$ = C, then $\overline{r}$ = C and the instruction is generated in C (fourth column).

Confidentiality - implicit indirect leak. Rule 4 prevents implicit indirect leaks (see §4). Without giving the implementation details, Rule 4 assigns a color to each basic block.[4] When Privagic visits a conditional jump controlled by a C register, it assigns the color C to the basic blocks of the "if" and "then" branches (basic bloc $\mathcal{B}$ in Figure 4, see §4). It does not propagate C to the joining point of the "if" (e.g., $\mathcal{C}$ in Figure 4) since this basic block does not carry sensitive information anymore.

Rule 4 ensures that if a basic block $\mathcal{B}$ has the color C, then the output register of any instruction that belongs to $\mathcal{B}$ has a color compatible with C. For example, in Figure 4, Privagic reports an error: x cannot take the blue color of $\mathcal{B}$ at line 5 since x is U (line 1).

Integrity. Privagic ensures that a store to a C memory location is necessarily executed in C (fourth column). Thanks to that, an enclave cannot modify the memory of another enclave.

Iago attacks. Privagic prevents Iago attacks by ensuring that a C instruction only consumes inputs generated by C or loaded from C. This is the case first because a C register is either loaded from C (Rule 1) or computed by an instruction generated in C (Rule 2). Moreover, since Privagic replicates the computation of the F register in each enclave, a F register is also generated in C (see §5.3). Since Rules 1 to 3 ensure that a C instruction can only consume C or F registers, and since the C and F registers are generated in C or loaded from C, we can conclude that a C instruction can only consume registers generated in C or loaded from C.

6.1.2 Relaxed mode. In relaxed mode, Privagic generates a F register when it loads a value from S (Rule 1). The relaxed mode enforces confidentiality because Rule 3 still ensures that a store cannot write a C register in a S memory location. The relaxed mode also enforces integrity because a store executed in C necessarily accesses a C memory location (Rule 3). However, the relaxed mode does not prevent Iago attacks. Using a F register in Rules 1 to 3 is not safe anymore since a F register can come from S, which can be poisoned by an attacker.

6.2 Direct call and entry points

Handling a function call is slightly more complex because the same function can be called from two sites with arguments with different colors. For this reason, when Privagic visits a call to a local function, i.e., a function for which the IR is available in the analyzed file, Privagic generates a specialized version of the function with the actual colors of the arguments. Privagic then visits the generated function, which means that, when Privagic visits a function, the colors of the arguments are always known.

To start an analysis pass, Privagic also generates specialized versions of what we name the *entry points*. An entry point is a function that may be called from another project. The arguments of an entry point are U in hardened mode, and F in relaxed mode.

[3] Note that the 4th confidentiality rule in §4 is enforced by the $\overline{*p} \sim \overline{p}$ of Rule 1.

[4] A basic block is a sequence of instructions without a jump and that do not contain instructions that are the target of a jump, except for the first and the last instruction [1].

In the stabilizing algorithm (see §5.2), a pass consists of analyzing the IR by starting from the entry points.

To handle the case where Privagic generates a library, Privagic considers by default that any function with the `extern` attribute is an entry point. The developer can also explicitly give the entry points by using annotations (e.g., only the `main` function).

6.3 Other function calls

If a function is external, i.e., if the analyzed IR file does not contain the code of the function, Privagic considers by default that the function belongs to the untrusted part of the application. For this reason, Privagic considers that the arguments have to be compatible with U.

An external function may be available from within the enclave. This is the case of the functions of the mini-libc library provided by Intel SDK. This library provides basic functions such as `memcpy` or `malloc` and is embedded by Privagic in each enclave. For such a function, the developer can annotate it with the `within` annotation. Privagic ensures that if a call to a within function takes an argument with the color C where C ≠ F, then the call is executed in C. Additionally, Privagic ensures that all the other arguments are compatible with C. Moreover, if the function takes pointers as an argument, Privagic ensures that the pointed value is compatible with C, which ensures that an enclave cannot let a sensitive value escape through a pointer to U during an external call.

Handling an indirect call is more complex because Privagic cannot always identify the called function during compilation. Because of that, Privagic uses a conservative approach. Privagic considers an indirect call as a direct call to an external function located in the untrusted part of the application. As for any external call, Privagic verifies that the arguments of an indirect call are compatible with U. Accordingly, when an instruction loads a function pointer, Privagic loads a pointer to a version of the function specialized for U arguments.

6.4 Communication with the outside

In hardened mode, an enclave is totally isolated from the rest of the system. To allow the enclave to communicate with the outside, Privagic provides the `ignore` annotation. A function with the `ignore` annotation behaves as a function with the `within` annotation, which means that as soon as one of the arguments is C with C ≠ F, the call is executed in the enclave C. However, in that case, instead of reporting an error if an argument (or the value pointed by an argument) is incompatible with C, Privagic ignores the argument.

Thanks to the `ignore` annotation, the developer can classify and declassify values. For example, the developer can annotate with `ignore` the `encrypt(plain, len, key, iv, cypher)` function of the `libssl` library. In this case, since Privagic allows the code to call `encrypt` with a C pointer for `plain` and a U pointer for `cypher`, the developer can use `encrypt` to declassify the encrypted result generated in `cypher`.

7 APPLICATION PARTITIONING

After having computed the colors, Privagic partitions the application by rewriting it. It rewrites the application in three steps: it rewrites the global variables, the multi-color structures, and finally the functions.

7.1 Global variables

As a first step, Privagic rewrites the global variables. For the variables that are not S, Privagic places them in their enclaves. Since a S variable can be used by any enclave in relaxed mode, a S variable has to be accessible from any enclave. However, an enclave is a shared library and it cannot use a symbol defined in the untrusted part of the application, which is the case of a S variable. For this reason, Privagic gathers all the S variables in a shared data structure stored in unsafe memory and replaces accordingly all the accesses to the S variables by accesses to this structure. When Privagic starts an enclave, it gives a pointer to this structure to the enclave, which allows the enclave to access the S variables without relying on symbol resolution.

7.2 Multi-color structures

As a second step, Privagic handles the structures with multiple colors, such as the data structure presented in Figure 1 (see §1).[5] For a multi-color data structure, Privagic cannot keep the fields packed in memory because an enclave is contiguous in the virtual address space. For that reason, Privagic introduces an indirection level: instead of storing directly the colored values in the structure, Privagic stores pointers to the colored values. In order to introduce this indirection, Privagic first analyzes the code to associate each allocation site (e.g., a call to `malloc`) to a data structure. Then, for each multi-color data structure, Privagic rewrites its allocation site to allocate the structure in unsafe memory and the colored fields in their enclave. Finally, Privagic rewrites the accesses to the fields to replace direct access with indirect access. For example, a `memcpy(&s->f, ...)` becomes a `memcpy(&s->ind->f, ...)`.

7.3 Code rewriting

Finally, Privagic partitions the code itself. While it partitions the code, Privagic generates a parallel code by leveraging the fact that two instructions with different colors do not have data dependencies. For that, Privagic supposes that the Privagic runtime runs a *worker thread* in each enclave for each application thread. The remainder of the section explains how Privagic partitions the code step by step.

7.3.1 Color set and chunks. To generate a parallel code, Privagic first computes the *color set* of each function. The color set of a function is the set of colors used by a function, F excluded. Figure 6 shows an example. In this example, the color set of `main` (line 4 in Figure 6) is equal to {*blue*, U} because `main` uses the U color at line 5, and the blue color at line 6. For the specialized version of `f` that receives a blue parameter (line 9), its color set is {*blue*} because `f` receives a blue argument, calls a function without using color, and returns the F value 42. For `g`, its color set is {*red*, *blue*, U} because it uses the red and blue colors at lines 14 and 15, and then executes a call to an external function, which is colored in U during the type analysis phase.

[5] Note that Privagic handles a bitfield because, in LLVM, a bitfield is translated into a normal field with a specific number of bits.

Global variables

```
1. int color(U) unsafe = 0;
2. int color(blue) blue = 10;
3. int color(red) red = 0;
```

Function `main` — *colorset* = {*blue*, U}

```
4. int main() {
5.   unsafe = 1;
6.   int x = f(blue);
7.   return x;
8. }
```

Function `f` — *colorset* = {*blue*}

```
9. int f(int y) {
10.   g(21);
11.   return 42;
12. }
```

Function `g` — *colorset* = {*red*, *blue*, U}

```
13. void g(int n) {
14.   blue = n;
15.   red = n;
16.   printf("Hello\n");
17. }
```

Figure 6: A complete example.

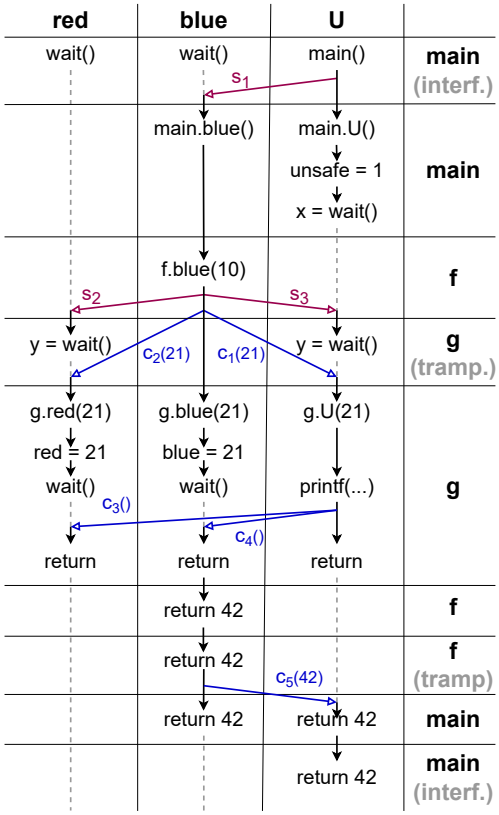

Figure 7: Execution of the example given in Figure 6.

For each color of a color set, Privagic generates a colored version of the function, which we call a *chunk*. For example, as shown in Figure 7, Privagic generates three chunks for the function g. Privagic partitions the code by generating the C instructions of a function in its C chunks.[6] Privagic also replicates the F instructions in each chunk, which ensures that using a F value in a chunk in hardened mode is safe (see §6.1.2). If the F instruction is uselessly replicated, a dead-code-elimination pass [1] eliminates it after.

7.3.2 Function call. For each call site, Privagic compares the color set of the caller with the color set of the callee. If a color C is common between the two color sets, Privagic generates a call to the chunk C of the callee directly in the chunk C of the caller. In that case, the chunk of the caller calls the chunk of the callee with the C and F arguments, but not with the others arguments. For example, in Figure 7, `main.blue` directly calls `f.blue` with the blue argument (10).

[6] As a corner case, for a store in S in a function without a S chunk, Privagic generates the store in an existing chunk to avoid uselessly creating a S chunk.

To handle the case where one of the colors of the callee is not in the color set of the caller, the Privagic runtime manages communication channels between the enclaves. In detail, each worker thread has a communication channel implemented as a lock-free FIFO queue [21, 28] stored in unsafe memory. To start a missing chunk, the Privagic runtime provides the `spawn` message, which takes as argument the identifier of the missing chunk.

The missing chunk may need F arguments, which are only computed in the caller. In this case, Privagic reports an error in hardened mode since using a value computed by another enclave is unsafe. In relaxed mode, using an unsafe value is authorized, and the Privagic runtime provides thus the `cont` message to send a F argument, and the `wait` function to wait for a message. In detail, for the F arguments, Privagic generates a trampoline in charge of receiving them from the caller, and then of calling the chunk. Figure 7 illustrates this case to start `g` from `f`. In this example, `f.blue` sends the spawn messages s_2 and s_3 in order to start the trampolines for the chunks `g.red` and `g.U`. Then, `f.blue` sends the F argument 21 with the `cont` messages c_1 and c_2. Finally, the trampolines in `g.red` and `g.U` receive the argument 21 and call the chunks.

7.3.3 Synchronization barriers. Some instructions have a visible effect and have thus to be executed in the sequential order of the source code. This is the case of a store to S and of a call to an external function (e.g., `printf`). When Privagic generates the code of an instruction that has a visible effect, Privagic generates a synchronization barrier by using the `cont` message and the `wait` function. For example, for the `printf` of `g` at line 16 in Figure 6, as shown in Figure 7, Privagic generates a synchronization barrier.

7.3.4 Entry points and indirect calls. For each entry point and each function indirectly called, Privagic considers that it executes only in U. For that reason, Privagic generates an *interface version* of these functions. An interface version keeps the original name of the function. It is in charge of starting the missing chunks. For example, in Figure 7, the interface function of `main` starts the execution of `main.blue` by sending a `spawn` message to the blue enclave and then directly calls `main.U`.

8 DISCUSSIONS AND LIMITATIONS

With Privagic, an attacker may try to attack an enclave by generating `cont` and `spawn` messages. Because a `cont` message simply unblocks an enclave without changing its execution flow, an attacker cannot temper the execution flow of an enclave with `cont` messages. In hardened mode, a `cont` message cannot carry a F value. This ensures that the enclave executes exactly the code that it is supposed to execute, even if an attacker sends unexpected `cont` messages. In relaxed mode, when a `cont` message carries a F

value, as for any value that comes from the untrusted part of the application, the developer may have to check its integrity.

An attacker can temper the execution flow of the application by sending unexpected `spawn` messages. To protect Privagic against this attack vector, identifying the valid sequences of `spawn` messages is required, which we let as future work.

A limitation of our prototype comes from the number of worker threads. Currently, for each thread of the application, Privagic runs one worker thread per enclave, which multiplies the number of threads by the number of colors plus one. We did not optimize this part of Privagic, but techniques such as configless switchless calls [48] should allow Privagic to adapt the number of worker threads to the workload.

Another limitation of our prototype comes from the impossibility of using multi-color structures in hardened mode. Because of the indirection introduced between the structure located in unsafe memory and the colored fields (see §7.2), an enclave has to load a pointer to a colored field from unsafe memory, which requires the use of the relaxed mode. Currently, in hardened mode, a developer can thus only use multiple colors if they do not belong to the same data structure. Using multi-color structures in hardened mode requires the use of authenticated pointers, which we let as a future work. Note that this restriction does not exist with a single color and that our goal is not to use multiple colors, but to partition multi-threaded applications.

9 EVALUATION

Explicit secure typing makes the partitioning of a multi-threaded application possible. However, it requires more engineering effort than data flow analysis. The first and main goal of our evaluation is thus to verify that this effort remains modest. Additionally, as for any tool that automatically partitions an application, Privagic is supposed to decrease the trusted computing base without degrading the performance too much. The second goal of our evaluation is thus to verify that this is the case. To evaluate the engineering effort, the trusted computing base, and the performance, we consider a large application (memcached) and several common data structures found in many applications.

9.1 Hardware and software setting

We evaluate Privagic on two machines. Machine A is an Intel i5-9500 CPU 3 GHz with 16 GiB memory. This 6-core CPU ships SGX version 1 with a maximal EPC size usable by the enclaves of 93 MiB. Machine B is an Intel Xeon Gold 5415+ with 16 CPUs, 120 GiB of memory, and 22.5 MiB of last-level cache. This processor supports SGX version 2 with a maximum EPC size of 8131 MiB.

Machines A and B run Linux 5.15.0, glibc 2.31, clang 10.0.0, and Intel SGX SDK 2.19.100. We run at least 5 times the applications and report the average and the variance.

9.2 Memcached

We first evaluate a legacy application: memcached 1.6.12 (24 841 lines of C code). Memcached is an in-memory cache widely used in production. It is designed as an event-based system with multiple threads to handle the requests. We inject the load with the standard Java version of the YCSB benchmark [15] through the loopback network on the same machine. YCSB simulates 6 clients with 6 threads. YCSB uses a record size of 1024 B and runs 8, 000, 000 operations. We evaluate data sets ranging from 1 MiB to 32 GiB by changing the number of records.

	Modified (C locs)	TCB (KiB)	User code (LLVM)
Scone	0	51271	78106 + libraries
Privagic	9	268	1 238

Table 4: Memcached metrics (locs: lines of code).

We configure memcached with 7 threads: a worker thread, a network listener thread, and some miscellaneous background threads, e.g., in charge of maintaining the least recently used key/value pairs in memory.[7]

We evaluate three configurations of memcached on machine B. Unprotected is an unprotected memcached running in a docker container. Scone relies on Scone [5] v5.7.0 to run memcached in a container fully embedded in an SGX enclave. Scone calls the operating system by using switchless calls [5]. Privagic is our version of memcached. It prevents an attacker from reading or writing the key/value pairs of memcached by coloring the central map of memcached. Privagic is generated in hardened mode. For a get operation, Privagic declassifies the result as soon as it is retrieved from the map. Declassifying a value is not necessary for a put operation.

9.2.1 Engineering effort. Column "Modified" of Table 4 reports the number of modified lines of code. Scone does not require any modification since memcached is fully embedded in an enclave. For Privagic, we modified 9 lines of code: 2 to add the colors to the central map, and 7 to declassify the values. With Glamdring [23], the authors report only 2 modified lines of code for memcached. The engineering effort required to use Privagic is thus larger but remains modest and realistic for a complete application.[8]

9.2.2 TCB size. As shown in the "TCB" column of Table 4, the binary code loaded in the enclave with Privagic is roughly 200 times smaller than the binary code loaded with Scone. With Scone, this code includes memcached (349 KiB), the musl C library (14.7 MiB), and the OS library shipped with Scone (36.2 MiB). Any bug from any of these components may lead to a threat inside the enclave.

The 268 KiB of Privagic includes the Intel SDK runtime and the Privagic runtime. If we exclude this code, the user code generated by Privagic from the source code of the application contains 1238 lines of LLVM code (column "User code"). We cannot report the exact user code with Scone because Scone does not disclose its source code. However, even if we ignore the musl C library and the library OS of Scone, the code of memcached embedded in an enclave is already 63 times larger (78106 lines of LLVM code).

Overall, these results show that Privagic significantly reduces the TCB as compared to fully embedding the application in the enclave, and for a modest engineering effort.

9.2.3 Performance. Figure 8 reports the throughput of memcached on machine B.

[7] https://github.com/memcached/memcached/blob/master/doc/threads.txt

[8] Note that the code generated by Glamdring is correct in this case because memcached does not face the concurrency issue presented in Figure 3. However, in the general case, Glamdring cannot handle multiple threads because of the limitation of the Eva plugin of Frama-C [10].

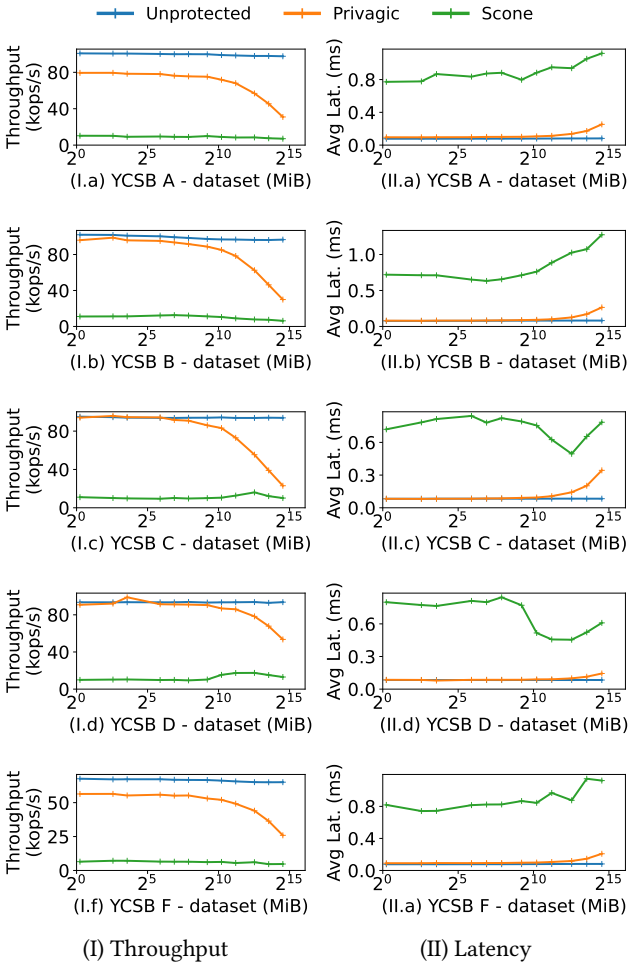

Figure 8: Memcached with YCSB.

Small data set. For a small dataset (less than 200 MiB), the throughput of Privagic is between 8.5 to 10.0 better than the throughput of Scone. The throughput of Privagic is only 5% to 20% lower than the throughput of Unprotected for a small dataset.

With Scone, the overhead is larger than with Privagic for two reasons. First, we measured that the time to enter and leave the enclave to execute a request is larger with Scone than with Privagic. Then, since Scone fully executes in an enclave, Scone has to perform many system calls from the enclave: for the network operations and to acquire/release locks. Even if Scone optimizes the system calls with switchless calls, the large number of system calls in Scone significantly slows down the execution of a request. This is not the case with Privagic because Privagic minimizes the code in the enclave. With Privagic, the code in the enclave accesses the data structure and only calls the operating system twice: to acquire a lock and to release it.

Large data set. We also observe that the latency and the throughput of Privagic degrades when the dataset size increases. In the worst case (dataset of 32 GiB), the throughput of Privagic remains, however, at least 2.3 times higher than the throughput of Scone.

The throughput of Privagic decreases with a large dataset because of cache effects. In detail, the central map of memcached becomes larger with a larger dataset. Retrieving a key thus leads to more memory accesses, which translates into more last-level cache (LLC) misses. This is the case for Unprotected, Privagic and Scone. For example, in Unprotected, we measured that increasing the dataset from 236 MiB to 32 GiB multiplies the ratio of LLC misses roughly by 3 (from 6.5% to 17.6% LLC misses).

For Privagic and Scone, the higher number of cache misses has an important effect because they happen while the processor is in enclave mode. As reported in [30], an LLC miss in enclave mode takes between 5.6 to 9.5 more time than in normal mode. As a result, while the higher number of LLC misses has a marginal effect for Unprotected, this is not the case for Privagic and Scone. For Privagic, the higher number of LLC misses translates into a worse latency, which itself translates into a degraded throughput. The throughput degradation with Scone also exists but is less visible because the latency of Scone is already high with a small dataset.

9.3 Data structures

To evaluate the engineering effort required to adapt an application for Privagic, we now focus on classical data structures found in many applications: a linked-list, a red-black tree (balanced tree) and a hashmap. The hashmap uses a separate chaining algorithm: it is designed as an array of linked lists, in which each linked list contains the keys that collide. We use the data structures as maps, i.e., they associate keys to values. The map is stored in a global variable, and we inject the load with our re-implementation in C of the YCSB benchmark [15]. The benchmark directly accesses the map in the same thread without involving the network in order to observe the cost of using SGX.

We evaluate five configurations. Unprotected does not use SGX. In Privagic-1, we color the whole data structure with Privagic in hardened mode. In Privagic-2, we use the relaxed mode of Privagic to color the keys and the values with two different colors. Intel-sdk-1 exposes the interface of the maps in EDL (i.e., `put`, `get`). Intel-sdk-2 uses two enclaves in EDL: one for the keys and the other for the values.

In each configuration, we use keys of 8 bytes and values of 1024 bytes. We use machine A for this evaluation. For the experiments with a single color, we pre-initialize the map with 100 000 keys and then run the experiment. For the experiments with two colors, we pre-initialize the map with only 20 000 keys because the runs are much longer.

9.3.1 Engineering effort. With one color, protecting the data structure with Privagic by starting from the unprotected version leads to the modification of at most 5 lines of code. For example, for the hashmap, we modified 1 line to color the data structure, 2 lines to color two local variables, and 2 lines to declassify the result of the get function. For two colors, we modified at most 6 lines in total. For example, for the hashmap, we modified 2 lines to add the colors to the fields, 1 line to color a local variable, 2 lines to declassify the result of a get, and 1 line to declassify the result of a call to a hash function.

Manually porting the unprotected code to use one color with Intel SDK is relatively straightforward, but leads, for example, to 206

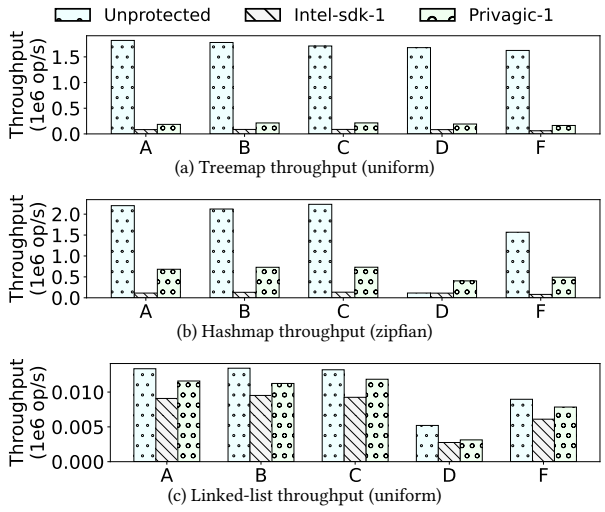

Figure 9: Data structures with YCSB (1 color).

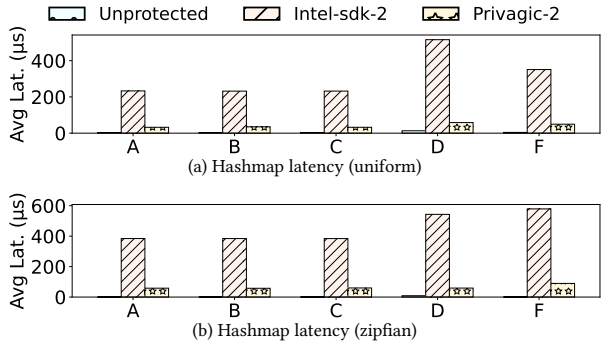

Figure 10: Hashmap with YCSB (2 colors).

modified lines of code for the hashmap. Manually porting the unprotected code to use two colors with Intel SDK is more challenging. To exchange data between the enclaves, we have to manually handle the allocations of the keys and values in the different enclaves, and to expose several functions of the enclaves to the untrusted part of the application. Overall, these modifications lead to a whole redesign of the code.

9.3.2 Performance analysis. Figure 9 reports the performance with one color, and Figure 10 with two colors.

In Figure 9, when we compare Intel-sdk-1 with Privagic-1, we observe that Privagic multiplies the throughput by 2.2 to 2.7 for the treemap, by 1.6 to 2.7 for the hashmap, and by 1.1 to 1.2 for the linked-list. The worse throughput of Intel-sdk-1 comes from a higher cost of crossing the enclave boundary: Privagic relies on a lock-free queue for communication while Intel-sdk-1 implements a switchless call with a lock [40, 43]. The performance improvement brought by Privagic is more interesting with the treemap and the hashmap than with the linked list because retrieving a key in a linked list requires visiting many (key, value) couples (50 000 in average), which amortizes the cost of crossing an enclave boundary.

Compared to Unprotected, Privagic-1 divides the throughput by 19.5 to 26.7 for the treemap, by 3.6 to 6.1 for the hashmap, and by 1.2 to 1.7 for the linked list. The degradation is more important with the treemap because of the uniform access pattern. This pattern leads to many LLC misses, which are 5.6 to 9.5 more costly in enclave mode [30]. The degradation is lower with the hashmap because of the zipfian access pattern, which leads to fewer LLC misses. In this case, the overhead comes from the cost of exchanging messages between the thread in normal mode and the thread in enclave mode. This cost dominates in this experiment because access to the hashmap only costs a few memory accesses. The overhead becomes lower with the linked list because the time to traverse the list is larger, and thus hides the cost of exchanging messages between the threads.

In Figure 10, we observe that Privagic divides the latency by 6.4 to 9.2 times. Two colors exacerbate the advantage of using Privagic compared to using Intel SDK because of more enclave transitions. We also observe that Privagic-2 significantly degrades latency compared to Unprotected for the same reason: Privagic-2 pays a large cost to cross multiple enclave boundaries for each request, while this cost does not exist with Unprotected.

9.4 Assessment

Overall, our evaluation shows that: (i) Privagic requires a modest engineering effort (at most 9 modified lines of code if we consider all our use cases), (ii) we can easily use Privagic with different data structures, (iii) Privagic is more efficient than using Intel SDK or than fully embedding the application in an enclave, and (iv) Privagic reduces the trusted computing base compared to fully embedding the application in an enclave.

10 CONCLUSION

This paper proposes explicit secure typing to ease the use of a TEE in a legacy application. Our evaluation shows that (i) explicit secure typing handles multiple threads in the general case, (ii) using explicit secure typing in a legacy application requires a modest engineering effort, and (iii) explicit secure typing reduces the TCB and is more efficient than embedding a whole application in an enclave.

ACKNOWLEDGMENT

We thank our shepherd, Prof. Roman Vitenberg, and the Middleware 2024 reviewers for their valuable efforts and suggestions. We also thank the reviewers from NDSS 2024, Eurosys 2024, and ICDCS 2024 for their comments that helped us to shape and polish the paper.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley, USA.
[2] Julien Amacher and Valerio Schiavoni. 2019. On the Performance of ARM TrustZone. In *Proceedings of the conference on Distributed Applications and Interoperable Systems, DAIS '19.* 133–151.
[3] Amazon. 2022. AWS nitro system. https://aws.amazon.com/ec2/nitro/
[4] Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language.* Ph. D. Dissertation. Johns Hopkins University.
[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, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In

Proceedings of the conference on Operating Systems Design and Implementation, OSDI '16. 689–703.

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the conference on Programming Language Design and Implementation, PLDI '14*. 259–269.

[7] Mike Barnett, Manuel Fahndrich, Francesco Logozzo, and Diego Garbervetsky. 2007. Annotations for (more) Precise Points-to Analysis. In *Proceedings of the International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*. 8.

[8] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the conference on Operating Systems Design and Implementation, OSDI '14*. 267–283.

[9] David Brumley and Dawn Song. 2004. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the USENIX Security Symposium '04*. 5.

[10] David Bühler, Pascal Cuoq, Boris Yakobowski, Matthieu Lemerre, André Maroneze, Valentin Perelle, and Virgile Prevosto. [n. d.]. Eva - The Evolved Value Analysis plug-in. http://frama-c.com/download/frama-c-eva-manual.pdf

[11] Stephen Checkoway and Hovav Shacham. 2013. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13*. 253–264.

[12] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. 2007. Secure Web Applications via Automatic Partitioning. In *Proceedings of the Symposium on Operating Systems Principles, SOSP '07*. 31–44.

[13] Google Cloud. 2022. Confidential computing. https://cloud.google.com/confidential-computing

[14] Confidential Computing Consortium. 2022. Confidential computing - open source community. https://confidentialcomputing.io/

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the Symposium on Cloud computing, SoCC '10*. 143–154.

[16] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* 2016 (2016), 86.

[17] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the symposium on Principles Of Programming Languages, POPL '77*. 238–252.

[18] Manuvir Das. 2000. Unification-Based Pointer Analysis with Directional Assignments. In *Proceedings of the conference on Programming Language Design and Implementation, PLDI '00*. 35–46.

[19] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (jul 1987), 319–349.

[20] Adrien Ghosn, James R. Larus, and Edouard Bugnion. 2019. Secured Routines: Language-based Construction of Trusted Execution Environments. In *Proceedings of the Usenix Annual Technical Conference, USENIX ATC '19*. 571–586.

[21] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint* (1st ed.). Morgan Kaufmann.

[22] David Kaplan, Jeremy Powell, and Tom Woller. 2021. *AMD Memory Encryption - Whitepaper v9*. Technical Report. AMD. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[23] 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. [n. d.]. Glamdring: Automatic Application Partitioning for Intel SGX. In *Proceedings of the Usenix Annual Technical Conference, USENIX ATC '17*. 285–298.

[24] Shen Liu, Gang Tan, and Trent Jaeger. 2017. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In *Proceedings of the ACM conference on Computer and Communications Security, CCS '17*. 2359–2371.

[25] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation. In *Proceedings of the ACM conference on Computer and Communications Security, CCS '15*. 1607–1619.

[26] Andrea Mambretti, Kaan Onarlioglu, Collin Mulliner, William Robertson, Engin Kirda, Federico Maggi, and Stefano Zanero. [n. d.]. Trellis: Privilege Separation for Multi-User Applications Made Easy. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 437–456.

[27] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2021. Twine: An Embedded Trusted Runtime for WebAssembly. In *Proceedings of the International Conference on Data Engineering, ICDE '21*. 205–216.

[28] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.* 15, 6 (jun 2004), 491–504.

[29] Microsoft. 2022. Microsoft Azure confidential computing. https://azure.microsoft.com/en-gb/solutions/confidential-compute/

[30] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the EuroSys European Conference on Computer Systems, EuroSys '17*. 238–253.

[31] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A Comprehensive Survey. *Comput. Surveys* 51 (01 2019), 1–36.

[32] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. 2019. SGX-LKL: Securing the host OS interface for trusted execution. *arXiv preprint arXiv:1908.11143* (2019).

[33] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. 2001. Points-to Analysis for Java Using Annotated Constraints. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '21*. 43–55.

[34] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. 2016. Automated Partitioning of Android Applications for Trusted Execution Environments. In *Proceedings of the International Conference on Software Engineering, ICSE '16*. 923–934.

[35] J.H. Saltzer and M.D. Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975), 1278–1308.

[36] Vasily A. Sartakov, Stefan Brenner, Sonia Ben Mokhtar, Sara Bouchenak, Gaël Thomas, and Rüdiger Kapitza. 2018. EActors: Fast and Flexible Trusted Computing Using SGX. In *Proceedings of the International Conference on Middleware, Middleware '18*. 187–200.

[37] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *Proceedings of the Network and Distributed System Security symposium, NDSS '17*. 15.

[38] Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Proceedings of the symposium on Principles Of Programming Languages, POPL '96*. 32–41.

[39] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. 2018. VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '18*. 665–678.

[40] Subashiny Tanigassalame, Yohan Pipereau, Adam Chader, Jana Toljaga, and Gaël Thomas. 2024. FastSGX: A Message-passing based Runtime for SGX. In *Proceedings of the International Conference on Advanced Information Networking and Applications, AINA '24*. 12.

[41] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the Usenix Annual Technical Conference, USENIX ATC '17*. 645–658.

[42] Chia-Che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E. Porter. 2020. Civet: An Efficient Java Partitioning Framework for Hardware Enclaves. In *Proceedings of the USENIX Security Symposium '20*. 505–522.

[43] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *Proceedings of the International Symposium on Computer Architecture, ISCA '17*. 81–93.

[44] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize Once, Start Fast: Application Initialization at Build Time. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '19*. 29.

[45] Yongzheng Wu, Jun Sun, Yang Liu, and Jin Song Dong. 2013. Automatically Partition Software into Least Privilege Components Using Dynamic Data Dependency Analysis. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE '13*. 323–333.

[46] Peterson Yuhala, Hugo Guiroux, Jean-Pierre Lozi, Pascal Felber, Valerio Schiavoni, Alain Tchana, and Gaël Thomas. 2023. SecV: Secure Code Partitioning via Multi-Language Secure Values. In *Proceedings of the International Conference on Middleware, Middleware '23*. ACM.

[47] Peterson Yuhala, Jämes Ménétrey, Pascal Felber, Valerio Schiavoni, Alain Tchana, Gaël Thomas, Hugo Guiroux, and Jean-Pierre Lozi. 2021. Montsalvat: Intel SGX Shielding for GraalVM Native Images. In *Proceedings of the International Conference on Middleware, Middleware '21*. 352–364.

[48] Peterson Yuhala, Michael Paper, Timothée Zerbib, Pascal Felber, Valerio Schiavoni, and Alain Tchana. 2023. SGX Switchless Calls Made Configless. In *Proceedings of the international conference on Dependable Systems and Networks, DSN '23*. 229–238.

[49] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. 2002. Secure Program Partitioning. *ACM Trans. Comput. Syst.* 20, 3 (aug 2002), 283–328.

[50] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. 2003. Using Replication and Partitioning to Build Secure Distributed Systems. In *Proceedings of the IEEE Symposium on Security and Privacy, SSP '03*. 236.