Multi-variant Execution at the Edge

Javier Cabrera Arteaga, Pierre Laperdrix, Martin Monperrus, and Benoit Baudry

Abstract—Edge-Cloud computing offloads parts of the computations that traditionally occurs in the cloud to edge nodes. The binary format WebAssembly is increasingly used to distribute and deploy services on such platforms. Edge-Cloud computing providers let their clients deploy stateless services in the form of WebAssembly binaries, which are then translated to machine code, sandboxed and executed at the edge. In this context, we propose a technique that (i) automatically diversifies WebAssembly binaries that are deployed to the edge and (ii) randomizes execution paths at runtime. Thus, an attacker cannot exploit all edge nodes with the same payload. Given a service, we automatically synthesize functionally equivalent variants for the functions providing the service. All the variants are then wrapped into a single multivariant WebAssembly binary. When the service endpoint is executed, every time a function is invoked, one of its variants is randomly selected. We implement this technique in the MEWE tool and we validate it with 7 services for which MEWE generates multivariant binaries that embed hundreds of function variants. We execute the multivariant binaries on the world-wide edge platform provided by Fastly, as part as a research collaboration. We show that multivariant binaries exhibit a real diversity of execution traces across the whole edge platform distributed around the globe.

Index Terms—Diversification, Edge-Cloud computing, Multivariant execution, WebAssembly.

1 Introduction

Edge-Cloud computing distributes a part of the data and computation to edge nodes [1], [2]. Edge nodes are servers located in many countries and regions so that Internet resources get closer to the end users, in order to reduce latency and save bandwidth. Video and music streaming services, mobile games, as well as e-commerce and news sites leverage this new type of cloud architecture to increase the quality of their services. For example, the New York Times website was able to serve more than 2 million concurrent visitors during the 2016 US presidential election with no difficulty thanks to Edge computing [3].

The state of the art of edge computing platforms like Cloudflare or Fastly use the binary format WebAssembly (aka Wasm) [4], [5] to deploy and execute on edge nodes. WebAssembly is a portable bytecode format designed to be lightweight, fast and safe [6], [7]. After compiling code to a WebAssembly binary, developers spawn an edge-enabled compute service by deploying the binary on all nodes in an Edge platform. Thanks to its simple memory and computation model, WebAssembly is considered safe [8], yet is not exempt of vulnerabilities either at the execution engine's level [9] or the binary itself [10]. Implementations in both, browsers and standalone runtimes [11], have been found to be vulnerable [10], [11], opening the door to different attacks. This means that if one node in an Edge network is vulnerable, all the others are vulnerable in the exact same manner as the same binary is replicated on each node. In other words, the same attacker payload would break all edge nodes at once. This illustrates how Edge computing is fragile with respect to systemic vulnerabilities for the whole network, like it happened on June 8, 2021 for Fastly [12].

In this work, we introduce Multivariant Execution for WebAssembly in the Edge(MEWE), a framework that generates diversified WebAssembly binaries so that no two executions in the edge network are identical. Our solution is inspired by N-variant systems [13] where diverse variants are assembled for secretless security. Here, our goal is to drastically increase the effort for exploitation through large-scale execution path randomization. MEWE operates in two distinct steps. At compile time, MEWE generates *variants* for different functions in the program. A function variant is semantically identical to the original function but structurally different, i.e., binary instructions are in different orders or have been replaced with equivalent ones. All the function variants for one service are then embedded in a single multivariant WebAssembly binary. At runtime, every time a function is invoked, one of its variant is randomly selected. This way, the actual execution path taken to provide the service is randomized each time the service is executed resulting in harder break-once-break-everywhere attacks.

We experiment MEWE with 7 services, composed of hundreds of functions. We successfully synthesize thousands of function variants, which create orders of magnitude more possible execution paths than in the original service. To determine if these new paths embedded in the service binaries are actually randomly triggered at runtime, we deploy and run them on the Fastly edge computing platform, a leading world-wide content-delivery network (CDN). We collaborated with Fastly to experiment MEWE on the actual production edge computing nodes that they provide to their clients. This means that all our experiments ran in a real-world setting, together with Fastly's customer applications, such as the New-York Times services mentioned earlier. For this experiment, we execute each multivariant binary thousands of times on every edge computing node provided by Fastly. This experiment shows that the multivariant binaries render the same service as the original, yet with highly diverse execution

J. Cabrera Arteaga is with the KTH Royal Institute of Technology, Stockholm, Sweden. E-mail: javierca@kth.se
P. Laperdrix. is with The French National Centre for Scientific Research (CNRS), France. E-mail: pierre.laperdrix@inria.fr
M. Monperrus is with the KTH Royal Institute of Technology, Stockholm, Sweden. E-mail: monperrus@kth.se
B. Baudry is with the KTH Royal Institute of Technology, Stockholm, Sweden. E-mail: baudry.kth.se

traces.

The novelty of our contribution is as follows. First, we are the first to perform software diversification in the context of edge computing, with experiments performed on a real-world, large-scale, commercial edge computing platform (Fastly). Second, very few works have looked at software diversity for WebAssembly [14], [11], our paper contributes to proving the feasibility of this challenging endeavour.

To sum up, our contributions are:

- MEWE: a framework that builds multivariant WebAssembly binaries for edge computing, combining the automatic synthesis of semantically equivalent function variants, with execution path randomization.
- Original results on the large-scale diversification of WebAssembly binaries, at the function and execution path levels.
- Empirical evidence of the feasibility of deploying our novel multivariant execution scheme on a real-world edge-computing platform.
- A publicly available prototype system, shared for future research on the topic: https://github.com/Jacarte/MEWE.

This work is structured as follows. First, Section 2 present a background on WebAssembly and its usage in an edge-cloud computing scenario. Section 3 introduces the architecture and foundation of MEWE while Section 4 and Section 5 present the different experiments we conducted to show the feasibility of our approach. Section 6 details the Related Work, Section 7 discusses our results while Section 8 concludes this paper.

2 Background

In this section we introduce WebAssembly, as well as the deployment model that edge-cloud platforms such as Fastly provide to their clients. This forms the technical context for our work.

2.1 WebAssembly

WebAssembly is a bytecode designed to bring safe, fast, portable and compact low-level code on the Web. The language was first publicly announced in 2015 and formalized by Haas et al. [6]. Since then, most major web browsers have implemented support for the standard. Besides the Web, WebAssembly is independent of any specific hardware, which means that it can run in standalone mode. This allows for the adoption of WebAssembly outside web browsers [7], e.g., for edge computing [11].

WebAssembly binaries are usually compiled from source code like C/C++ or Rust. Listing 1 and 2 illustrate an example of a C function turned into WebAssembly. Listing 1 presents the C code of one function and Listing 2 shows the result of compiling this function into a WebAssembly module. The WebAssembly code is further interpreted or compiled ahead of time into machine code.

2.2 Web Assembly and Edge Computing

Using Wasm as an intermediate layer is better in terms of startup and memory usage, than containerization or virtualization [8], [15]. This has encouraged edge computing platforms like Cloudflare or Fastly to adopt WebAssembly

```c
int f(int x) {
    return 2 * x + x;
}
```

Listing 1: C function that calculates the quantity $2x + x$

```
(module
  (type (;0;) (func (param i32) (result i32)))
  (func (;0;) (type 0) (param i32) (result i32)
    local.get 0
    local.get 0
    i32.const 2
    i32.mul
    i32.add)
    (export "f" (func 0)))
```

Listing 2: WebAssembly code for Listing 1.

(Wasm) to deploy client applications in a modular and sandboxed manner [4], [5]. In addition, WebAssembly is a compact representation of code, which saves bandwidth when transporting code over the network . This allows edge-cloud platform providers to deploy the same Wasm binary, for a client application, around the world in a few seconds.

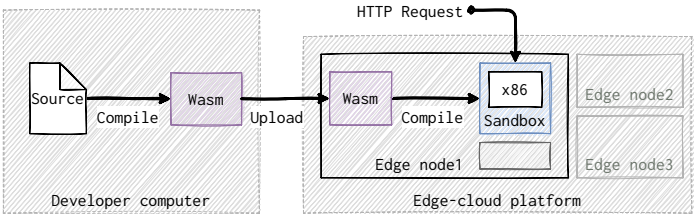

Fig. 1: Deployment to the edge. Client application developers build their application and compile it to WebAssembly before submitting to the Edge-Cloud computing platform. Then, the Wasm binary is distributed to all edge nodes, where it is compiled to a sandboxed machine code (x86 or ARM). This machine code is executed at every service request.

Client applications that are designed to be deployed on edge-cloud computing platforms are usually isolated services, having one single responsibility. This development model is known as serverless computing, or function-as-a-service [16], [11]. Figure 1 summarizes the development and deployment process for a client application to an edge-cloud platform. First, the developers of a client application implement the isolated services in a given programming language. The source code and the HTTP harness are then compiled to WebAssembly using a tool like LLVM [17] or Binaryen [18].

When client application developers deploy a WebAssembly binary for a function-as-a-service, it is sent to all edge nodes in the platform. Then, the WebAssembly binary is compiled on each node to machine code. This way, if the edge nodes have different architectures, the clients can still deploy a single WebAssembly binary, and the compilers take care of the final machine code generation step. Each binary is compiled in a way that ensures that the code runs inside an isolated sandbox.

2.3 Multivariant Execution

In 2006, security researchers of University of Virginia have laid the foundations of a novel approach to security that

consists in executing multiple variants of the same program. They called this "N-variant systems" [13]. This potent idea has been renamed soon after as "multivariant execution".

Definition 1. Multivariant execution (MVE) consists of creating multiple variants of a program and executing them in a principled way so as to improve security and reliability.

There is a wide range of realizations of MVE in different contexts. Bruschi et al. [19] and Salamat et al. [20] pioneered the idea of executing the variants in parallel. Subsequent techniques focus on MVE for mitigating memory vulnerabilities [21], [22] and other specific security problems incl. return-oriented programming attacks [23] and code injection [24]. A key design decision of MVE is whether it is achieved in kernel space [25], in user-space [26], with exploiting hardware features [27], or even throught code polymorphism [28]. Finally, one can neatly exploit the limit case of executing only two variants [29], [30]. The body of research on MVE in a distributed setting has been less researched. Notably, Voulimeneas et al. proposed a multivariant execution system by parallelizing the execution of the variants in different machines [31] for sake of efficiency.

In this paper, we propose an original kind of MVE in the context of edge computing. We generate multiple program variants, which we execute on edge computing nodes. We use the natural redundancy of Edge-Cloud computing architectures to deploy an internet-based MVE. Next section goes into the details of our procedure to generate variants and assemble them into multivariant binaries.

3 MEWE: Multivariant Execution for Edge Computing

In this section we present MEWE, a novel technique to synthesize multivariant binaries and deploy them on an edge computing platform.

3.1 Overview

The goal of MEWE is to synthesize multivariant WebAssembly binaries, according to the threat model presented in Section 3.2.1. The tool generates application-level multivariant binaries, without any change to the operating system or WebAssembly runtime. The core idea of MEWE is to synthesize diversified function variants providing execution-path randomization, according to the diversity model presented in Section 3.2.3.

In Figure 2, we summarize the analysis and transformation pipeline of MEWE. We pass a bitcode to be diversified, as an input to MEWE. Analysis and transformations are performed at the level of LLVM's intermediate representation (LLVM IR), as it is the best format for us to perform our modifications (see Section 3.2.4). LLVM binaries can be obtained from any language with an LLVM frontend such as C/C++, Rust or Go, and they can easily be compiled to WebAssembly. In Step ①, the binary is passed to CROW [14], which is a superdiversifier for Wasm that generates a set of variants for the functions in the binary. Step ② packages all the variants in one single multivariant LLVM binary. In Step ③, we use a special component, called a "mixer", which augments the binary with two different components: an HTTP endpoint harness and a random generator, which are both required for executing Wasm at the edge. The harness is used to connect the program to its execution environment while the generator provides support for random execution path at runtime. The final output of Step ④ is a standalone multivariant WebAssembly binary that can be deployed on an edge-cloud computing platform. In the following sections, we describe in greater details the different stages of the workflow.

3.2 Key design choices

In this section, we introduce the main design decisions behind MEWE, starting from the threat model, to aligning the code analysis and transformation techniques.

3.2.1 Threat Model

With MEWE, we aim to defend against an attacker that wants to leak private and sensitive information by learning about the state of the system and its characteristics. These attacks include but are not limited to timing specific operations [32], [33], counting register spill/reload operations to study and attack memory side-channels [34] and performing call stack analysis. They can be performed either locally or remotely by finding a vulnerability or using shared resources in the case of a multi-tenant Edge computing server but the details of such exploitation are out of scope of this study.

3.2.2 Dangers related to Edge Computing

To benefit from the performance improvements offered by edge computing, Fastly and Cloudflare modularize their services into a set of WebAssembly functions, which are deployed on all the edge nodes provided by the edge computing platforms. Then, these services are spawned on demand. This model of distributing the exact same WebAssembly binary on hundreds of computation nodes around the world is a serious risk for the infrastructure: a malicious developer who manages to exploit one vulnerability in the binary can exploit all the compute nodes with the same attack vector [35].

3.2.3 Execution Diversification Model

MEWE is designed to randomize the execution of WebAssembly programs, via diversification transformations. Per Crane et al. those transformations are made to hinder side-channel attacks [36]. All programs are diversified with behavior preservation guarantees [14]. The core diversification strategies are:

1) *Constant Inferring*. MEWE identifies variables whose value can computed at compile time. This has an effect on program execution times [37].
2) *Call Stack Randomization*. MEWE introduces equivalent synthetic functions that are called randomly. This results in randomized call stacks, which complicates attacks based on call stack analysis [38].
3) *Inline Expansion*. MEWE inlines methods when appropriate. This also results in different call stacks, to hinder the same kind of attacks as for call stack randomization [38]. On the other hand, the number of register spill/reload operations during machine code generation changes, affecting the measurement of memory side-channels [34].

3.2.4 Diversification at the LLVM level

MEWE diversifies programs at the LLVM level. Other solutions would have been to diversify at the source code level [39], or at the native binary level, eg x86 [40]. However, the former would limit the applicability of our work. The latter is not compatible with edge computing: the top edge computing execution platforms, e.g. Cloudflare and Fastly, mostly take WebAssembly binaries as input.

LLVM, on the contrary, does not suffer from those limitations: 1) it supports different languages, with a rich ecosystem of frontends 2) it can reliably be retargeted to WebAssembly, thanks to the corresponding mature component in the LLVM toolchain. In addition, the LLVM ecosystem as a whole is very active, providing us with many different tools to facilitate our research endeavour.

3.3 Variant generation

MEWE relies on the superdiversifier CROW [14] to automatically diversify each function in the input LLVM binary (Step ①). CROW receives an LLVM module, analyzes the binary at the function block level and generates semantically equivalent variants for each function, if they exist. CROW variants are verified as semantically equivalent with an SMT solver. Here, we define a function variant as:

Definition 2. Function variant: Let F be a function, F' is a function variant of F for MEWE if it is semantically equivalent (i.e., same input/output behavior), but exhibits a different internal behavior through tracing.

In Listing 3, we illustrate two semantically equivalent Wasm functions according to Definition 2. The left most listing corresponds to the Wasm module shown in Listing 2. The right most listing is a variant for this function. We can appreciate that the multiplication of the original code, in the third and four lines, is replaced by an addition, making the variant to have the same semantic but executing different instructions.

```
...
(func (;0;)
    local.get 0
    local.get 0
    i32.const 2
    i32.mul
    i32.add)
...
```

```
...
(func (;0;)
    local.get 0
    local.get 0
    local.get 0
    i32.add
    i32.add)
...
```

Listing 3: Example of two semantically equivalent functions. The left listing corresponds to the original code. The right listing shows a semantically equivalent variant.

CROW synthesizes variants by enumerative synthesis based on code transformation. The most relevant transformations are: constant inferring to replace control flow statements, arithmetic's equivalent replacement, and loop unrolling. CROW performs stacked transformations, this means that it can synthesize variants of different size, i.e., from smaller to larger variants than the original.

The variants created by CROW are artificially synthesized from the original binary. CROW checks for semantic equivalence of both codes, original and variant using the symbolic execution. If the behavior of the variant is not the reference behavior, the variant is discarded. This means that, after Step ①, the variant is necessarily equivalent to the original program.

3.4 Combining variants into multivariant functions

Step ② of MEWE consists in combining the variants generated for the original functions, into into a single binary. The goal is to support execution-path randomization at runtime. The core idea is to introduce one dispatcher function per original function for which we generate variants. A dispatcher function is a synthetic function that is in charge of choosing a variant at random, every time the original function is invoked during the execution. The random invocation of different variants at runtime is a known randomization technique, for example used by Lettner et al. with sanitizers [41].

With the introduction of dispatcher function, MEWE turns the original call graph into a multivariant call graph, defined as follows.

Definition 3. Multivariant Call Graph (MCG): A multivariant call graph is a call graph $\langle N, E \rangle$ where the nodes in N represent all the functions in the binary and an edge $(f_1, f_2) \in E$ represents a possible invocation of f_2 by f_1 [42], where the nodes are typed. The nodes in N have three possible types: a function present in the original program, a generated function variant, or a dispatcher function.

In Figure 3, we show the original static call graph for program bin2base64 (top of the figure), as well as the multivariant call graph generated with MEWE (bottom of the figure). The grey nodes represent function variants, the green nodes function dispatchers and the yellow nodes are the original functions. The possible calls are represented by the directed edges. The original bin2base64 includes 3 functions. MEWE generates 43 variants for the first function, none for the second and three for the third function. MEWE introduces two dispatcher nodes, for the first and third functions. Each dispatcher is connected to the corresponding function variants, in order to invoke one variant randomly at runtime.

In Listing 4, we illustrate the LLVM construction for the function dispatcher corresponding to the right most green node of Figure 3. It first calls the random generator, which returns a value that is then used to invoke a specific function variant. It should be noted that the dispatcher function is constructed using the same signature as the original function.

We implement the dispatchers with a switch-case structure to avoid indirect calls that can be susceptible to speculative execution based attacks [11]. The choice of a switch-case also avoids having multiple function definitions with the same signature, which could increase the attack surface in case the function signature is vulnerable [43]. This also allows MEWE to inline function variants inside the dispatcher, instead of defining them again. Here we trade security over performance, since dispatcher functions that perform indirect calls, instead of a switch-case, could improve the performance of the dispatchers as indirect calls have constant time.

3.5 MEWE's Mixer

The MEWE mixer has four specific objectives: wrap functions as HTTP endpoints, link the LLVM multivariant binary, inject a random generator and merge all these components into a multivariant WebAssembly binary.

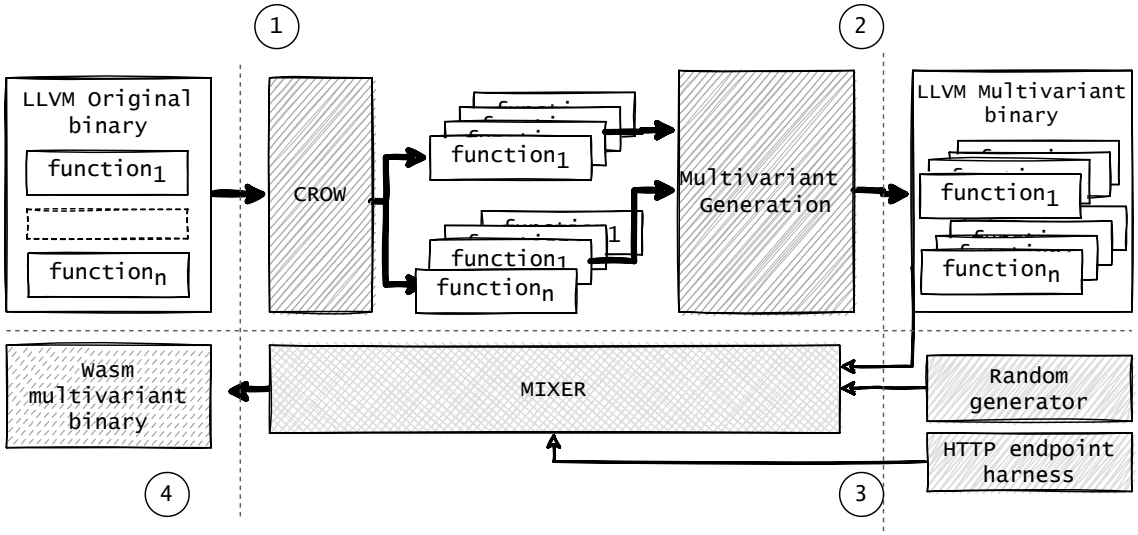

Fig. 2: Overview of MEWE. It takes as input the LLVM binary representation of a service composed of multiple functions. It first generates a set of functionally equivalent variants for each function in the binary and then generates a LLVM multivariant binary composed of all the function variants as well as dispatcher functions in charge of selecting a variant when a function is invoked. The MEWE mixer composes the LLVM multivariant binary with a random number generation library and an edge specific HTTP harness, in order to produce a WebAssembly multivariant binary accessible through an HTTP endpoint and ready to be deployed to the edge.

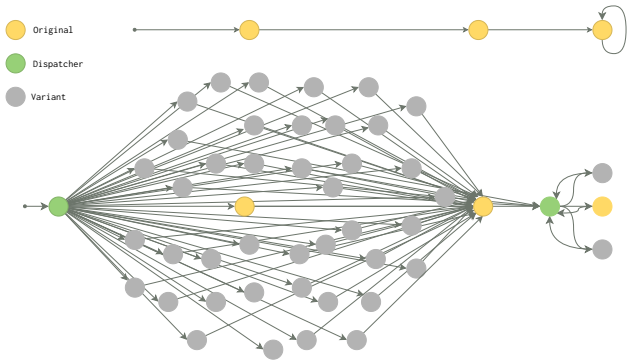

Fig. 3: Example of two static call graphs for the bin2base64 endpoint of libsodium. At the top, the original call graph, at the bottom, the multivariant call graph, which includes nodes that represent function variants (in grey), dispatchers (in green), and original functions (in yellow).

```
define internal i32 @b64_byte2urlsafe_char(i32 %0) {
  entry:
    %1 = call i32 @discriminate(i32 3)
    switch i32 %1, label %end [
      i32 0, label %case_43_
      i32 1, label %case_44_
    ]
  case_43_: ; preds = %entry
    %2 = call i32 @b64_byte_to_urlsafe_char_43_(%0)
    ret i32 %2
  case_44_: ; preds = %entry
    %3 = call i32 @b64_byte_to_urlsafe_char_44_(%0)
    ret i32 %3
  end: ; preds = %entry
    %4 = call i32 @b64_byte2urlsafe_char_original(%0)
    ret i32 %4
}
```

Listing 4: Dispatcher function embedded in the multivariant binary of the bin2base64 endpoint of libsodium, which corresponds to the rightmost green node in Figure 3.

We use the Rustc compiler[1] to orchestrate the mixing because Rustc is a compiler able to merge custom Rust source code with arbitratry-compatible LLVM binary producing a final Wasm binary. For the generator, we rely on WASI's specification [44] for the random behavior of the dispatchers. Its exact implementation is dependent on the platform on which the binary is deployed. For the HTTP harnesses, since our edge computing use case is based on the Fastly infrastructure, we rely on the Fastly API[2] to transform our Wasm binaries into HTTP endpoints. The harness enables a function to be called as an HTTP request and to return a HTTP response. Throughout this paper, we refer to an endpoint as the closure of invoked functions when the entry point of the WebAssembly binary is executed.

3.6 Multivariant Binary Execution at the Edge

When a WebAssembly binary is deployed on an edge platform, it is translated to machine code on the fly. For our experiment, we deploy on the production edge nodes of Fastly. This edge computing platform uses Lucet, a native WebAssembly compiler and runtime, to compile and run the deployed Wasm binary [3]. Lucet generates x86 machine code and ensures that the generated machine code executes inside a secure sandbox, controlling memory isolation.

Figure 4 illustrates the runtime behavior of the original and the multivariant binary, when deployed on an

1. https://doc.rust-lang.org/rustc/what-is-rustc.html
2. https://docs.rs/crate/fastly/0.7.3
3. https://github.com/bytecodealliance/lucet

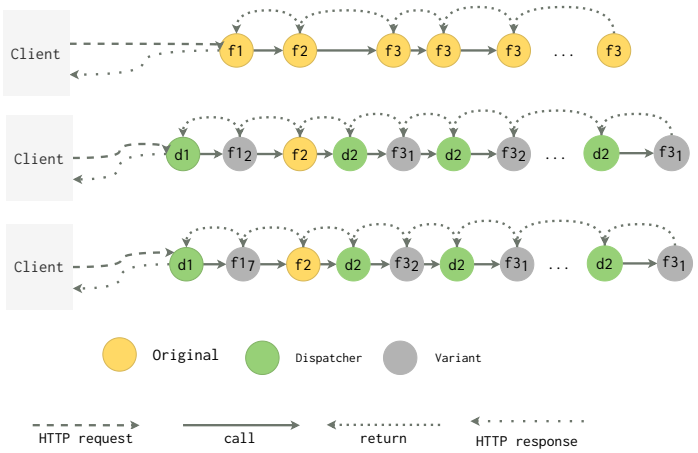

Fig. 4: Top: an execution trace for the `bin2base64` endpoint. Middle and bottom: two different execution traces for the multivariant `bin2base64`, exhibited by two different requests with exactly the same input.

Edge node. The top most diagram illustrates the execution trace for the original of the endpoint `bin2base64`. When the HTTP request with the input `"HelloWorld!"` is received, it invokes functions $f1$, $f2$ followed by 27 recursive calls of function $f3$. Then, the endpoint sends the result `"0x000xccv0x10x00b3Jsx130x000x00 0x00xpopAHRvdGE="` of its base64 encoding in an HTTP response.

The two diagrams at the bottom of Figure 4 illustrate two executions traces observed through two different requests to the endpoint `bin2base64`. In the first case, the request first triggers the invocation of dispatcher $d1$, which randomly decides to invoke the variant $f1_2$; then $f2$, which has not been diversified by MEWE, is invoked; then the recursive invocations to $f3$ are replaced by iterations over the execution of dispatcher $d2$ followed by a random choice of variants of $f3$. Eventually the result is computed and sent back as an HTTP response. The second execution trace of the multivariant binary shows the same sequence of dispatcher and function calls as the previous trace, and also shows that for a different requests, the variants of $f1$ and $f3$ are different.

The key insights from these figures are as follows. First, from a client's point of view, a request to the original or to a multivariant endpoint, is completely transparent. Clients send the same data, receive the same result, through the same protocol, in both cases. Second, this figure shows that, at runtime, the execution paths for the same endpoint are different from one execution to another, and that this randomization process results from multiple random choices among function variants, made through the execution of the endpoint.

3.7 Implementation

The multivariant combination (Step ②) is implemented in 942 lines of C++ code. Its uses the LLVM 12.0.0 libraries to extend the LLVM standard linker tool capability with the multivariant generation. MEWE's Mixer (Step ③) is implemented as an orchestration of the rustc and the WebAssembly backend provided by CROW. For sake of open science and for fostering research on this important topic, the code of MEWE is made publicly available on GitHub: https://github.com/Jacarte/MEWE.

4 Experimental Methodology

In this section we introduce our methodology to evaluate MEWE. First, we present our research questions and the services with which we experiment the generation and the execution of multivariant binaries. Then, we detail the methodology for each research question.

4.1 Research questions

To evaluate the capabilities of MEWE, we formulate the following research questions:

RQ1: **(Multivariant Generation) How much diversity can MEWE synthesize and embed in a multivariant binary ?** MEWE packages function variants in multivariant binaries. With this first question, we aim at measuring the amount of diversity that MEWE can synthesize in the call graph of a program.

RQ2: **(Intra MVE) To what extent does MEWE achieve multivariant executions on an edge compute node?** With this question we assess the ability of MEWE to produce binaries that actually exhibit random execution paths when executed on one edge node.

RQ3: **(Internet MVE) To what extent does MEWE achieve multivariant execution over the worldwide Fastly infrastructure?** We check the diversity of execution traces gathered from the execution of a multivariant binary. The traces are collected from all edge nodes in order to assess MVE at a worldwide scale.

RQ4: **What is the impact of the proposed multi-version execution on timing side-channels?** MEWE generates binaries that embed a multivariant behavior. We measure to what extent MEWE generates different execution times on the edge. Then, we discuss how multivariant binaries contribute to less predictable timing side-channels.

The core of the validation methodology for our tool MEWE, consists in building multivariant binaries for several, relevant endpoints and to deploy and execute them on the Fastly edge-cloud platform.

4.2 Study subjects

We select two mature and typical edge-cloud computing projects to study the feasibility of MEWE. The projects are selected based on: suitability for diversity synthesis with CROW (the projects should have the ability to collect their modules in LLVM intermediate representation), suitability for deployment on the Fastly infrastructure (the project should be easily portable Wasm/WASI and compatible with the Rust Fastly API), low chances to hit execution paths with no dispatchers and possibility to collect their execution runtime information (the endpoints should execute in a reasonable time of maximum 1 second even with the overhead of instrumentation). The selected projects are: **libsodium**, an encryption, decryption, signature and password hashing library which can be ported to WebAssembly and **qrcode-rust**, a QrCode and MicroQrCode generator written in Rust.

Name	#Endpoints	#Functions	#Instr.
libsodium https://github.com/jedisct1/libsodium	5	62	6187
qrcode-rust https://github.com/kennytm/qrcode-rust	2	1840	127700

TABLE 1: Selected projects to evaluate MEWE: project name; the number of endpoints in the project that we consider for our experiments, the total number of functions to implement the endpoints, and the total number of WebAssembly instructions in the original binaries.

In Table 1, we summarize some key metrics that capture the relevance of the selected projects. The table shows the project name with its repository address, the number of selected endpoints for which we build multivariant binaries, the total number of functions included in the endpoints and the total number of Wasm instructions in the original binary. Notice that, the metadata is extracted from the Wasm binaries before they are sent to the edge-cloud computing platform, thus, the number of functions might be not the same in the static analysis of the project source code

4.3 Experimentation platform

We run all our experiments on the Fastly edge computing platform. We deploy and execute the original and the multivariant endpoints on 64 edge nodes located around the world[4]. These edge nodes usually have an arbitrary and heterogeneous composition in terms of architecture and CPU model. The deployment procedure is the same as the one described in Section 2.2. The developers implement and compile their services to WebAssembly. In the case of Fastly, the WebAssembly binaries need to be implemented with the Fastly platform API specification so they can properly deal with HTTP requests. When the compiled binary is transmitted to Fastly, it is translated to x86 machine code with Lucet, which ensures the isolation of the service.

4.4 RQ1 Multivariant diversity

We run MEWE on each endpoint function of our 7 endpoints. In this experiment, we bound the search for function variant with timeout of 5 minutes per function. This produces one multivariant binary for each endpoint. To answer RQ1, we measure the number of function variants embedded in each multivariant binary, as well as the number of execution paths that are added in the mutivariant call graphs, thanks to the function variants.

4.5 RQ2 Intra MTD

We deploy the multivariant binaries of each of the 7 endpoints presented in Table 2, on the 64 edge nodes of Fastly. We execute each endpoint, multiple times on each node, to measure the diversity of execution traces that are exhibited by the multivariant binaries. We have a time budget of 48 hours for this experiment. Within this timeframe, we can query each endpoint 100 times on each node. Each query on the same endpoint is performed with the same input value. This is to guarantee that, if we observe different traces for different executions, it is due to the presence of multiple function variants. The input values are available as part of our reproduction package.

For each query, we collect the execution trace , i.e., the sequence of function names that have been executed when triggering the query. To observe these traces, we instrument the multivariant binaries to record each function entrance.

To answer RQ2, we measure the number of unique execution traces exhibited by each multivariant binary, on each separate edge node. To compare the traces, we hash them with the `sha256` function. We then calculate the number of unique hashes among the 100 traces collected for an endpoint on one edge node. We formulate the following definitions to construct the metric for RQ3.

Metric 1. Unique traces: $R(n, e)$. Let $S(n, e) = \{T_1, T_2, ..., T_{100}\}$ be the collection of 100 traces collected for one endpoint e on an edge node n, $H(n, e)$ the collection of hashes of each trace and $U(n, e)$ the set of unique trace hashes in $H(n, e)$. The uniqueness ratio of traces collected for edge node n and endpoint e is defined as

$$R(n, e) = \frac{|U(n, e)|}{|H(n, e)|}$$

The inputs that we pass to execute the endpoints at the edge and the received output for all executions are available in the reproduction repository at https://github.com/Jacarte/MEWE.

4.6 RQ3 Inter MTD

We answer RQ3 by calculating the normalized Shannon entropy for all collected execution traces for each endpoint. We define the following metric.

Metric 2. Normalized Shannon entropy: $E(e)$ Let e be an endpoint, $C(e) = \cdot_{n=0}^{64} H(n, e)$ be the union of all trace hashes for all edge nodes. The normalized Shannon Entropy for the endpoint e over the collected traces is defined as:

$$E(e) = -\Sigma \frac{p_x * log(p_x)}{log(|C(e)|)}$$

Where p_x is the discrete probability of the occurrence of the hash x over $C(e)$.

Notice that we normalize the standard definition of the Shannon Entropy by using the perfect case where all trace hashes are different. This normalization allows us to compare the calculated entropy between endpoints. The value of the metric can go from 0 to 1. The worst entropy, value 0, means that the endpoint always perform the same path independently of the edge node and the number of times the trace is collected for the same node. On the contrary, 1 for the best entropy, when each edge node executes a different path every time the endpoint is requested.

The Shannon Entropy gives the uncertainty in the outcome of a sampling process. If a specific trace has a high frequency

4. The number of nodes provided in the whole platform is 72, we decided to keep only the 64 nodes that remained stable during our experimentation.

of appearing in part of the sampling, then it is certain that this trace will appear in the other part of the sampling.

We calculate the metric for the 7 endpoints, for 100 traces collected from 64 edge nodes, for a total of 6400 collected traces per endpoint. Each trace is collected in a round robin strategy, i.e., the traces are collected from the 64 edge nodes sequentially. For example, we collect the first trace from all nodes before continuing to the collection of the second trace. This process is followed until 100 traces are collected from all edge nodes.

4.7 RQ4 Timing side-channels

For each endpoint listed in Table 2, we measure the impact of MEWE on timing. For this, we use the following metric:

Metric 3. Execution time: For a deployed binary on the edge, the execution time is the time spent on the edge to execute the binary.

Note that edge-computing platforms are, by definition, reached from the Internet. Consequently, there may be latency in the timing measurement due to round-trip HTTP requests. This can bias the distribution of measured execution times for the multivariant binary. To avoid this bias, we instrument the code to only measure the execution on the edge nodes.

We collect 100k execution times for each binary, both the original and multivariant binaries. We perform a Mann-Withney U test [45] to compare both execution time distributions. If the P-value is lower than 0.05, two compared distributions are different.

5 Experimental Results

5.1 RQ1 Results: Multivariant generation

We use MEWE to generate a multivariant binary for each of the 7 endpoints included in our 2 study subjects. We then calculate the number of diversified functions, in each endpoint, as well as how they combine to increase the number of possible execution paths in the static call graph for the original and the multivariant binaries.

The sections 'Original binary' and 'Multivariant WebAssembly binary' of Table 2 summarize the key data for RQ1. In the 'Original binary' section, the first column (#Functions) gives the number of functions in the original binary and the second column (#Paths) gives the number of possible execution paths in the original static call graph. The 'Multivariant WebAssembly binary' section first shows the number of each type of nodes in the multivariant call graph: #Non div. is the number of original functions that could not be diversified by MEWE, #Dispatchers is the number of dispatcher nodes generated by MEWE for each function that was successfully diversified, and #Variants is the total number of function variants generated by MEWE. The last column of this section is the number of possible execution paths in the static multivariant call graph.

For all 7 endpoints, MEWE was able to diversify several functions and to combine them in order to increase the number of possible execution paths in several orders of magnitude. For example, in the case of the `encrypt` function of libsodium, the original binary contains 23 functions that can be combined in 4 different paths. MEWE generated a total of 56 variants for 5 of the 23 functions. These variants, combined with the 18 original functions in the multivariant call graph, form 325 execution paths. In other words, the number of possible ways to achieve the same encryption function has increased from 4 to 325, including dispatcher nodes that are in charge of randomizing the choice of variants at 5 different locations of the call graph. This increased number of possible paths, combined with random choices, made at runtime, increases the effort a potential attacker needs to guess what variant is executed and hence what vulnerability she can exploit.

We have observed that there is no linear correlation between the number of diversified functions, the number of generated variants and the number of execution paths. We have manually analyzed the endpoint with the largest number of possible execution paths in the multivariant Wasm binary: `qr_str` of qrcode-rust. MEWE generated 2092 function variants for this endpoint. Moreover, MEWE inserted 17 dispatchers in the call graph of the endpoint. For each dispatcher, MEWE includes between 428 and 3 variants. If the original execution path contains function for which MEWE is able to generate variants, then, there is a combinatorial explosion in the number of execution paths for the generated Wasm multivariant module. The increase of the possible execution paths theoretically augments the uncertainty on which one to perform, in the latter case, approx. 140 000 times. As Cabrera and colleagues observed [14] for CROW, a large presence of loops and arithmetic operations in the original function code leverages to more diversification.

Looking at the #Dispatchers and #Variants columns of the 'Multivariant WebAssembly binary' section of Table 2, we notice that the number of variants generated per function greatly varies. For example, for both the `invert` and the `bin2base64` functions of Libsodium, MEWE manages to diversify 2 functions (reflected by the presence of 2 dispatcher nodes in the multivariant call graph). Yet, MEWE generates a total of 125 variants for the 2 functions in `invert`, and only 47 variants for the 2 functions in `bin2base64`. The main reason for this is related to the complexity of the diversified functions, which impacts the opportunities for the synthesis of code variations.

Columns #Originals of the 'Multivariant WebAssembly binary' section of Table 2 indicates that, in each endpoint, there exists a number of functions for which MEWE did not manage to generate variants. We identify three reasons for this, related to the diversification procedure of CROW, used by MEWE to diversify individual functions. First, some functions cannot be diversified by CROW, e.g., functions that wrap only memory operations, which are oblivious to CROW diversification technique. Second, the complexity of the function directly affects the number of variants that CROW can generate. Third, the diversification procedure of CROW is essentially a search procedure, which results are directly impacted by the tie budget for the search. In all experiments we give CROW 5 minutes maximum to synthesize function variants, which is a low budget for many functions. It is important to notice that, the successful diversification of some functions in each endpoint, and their combination within the call graph of the endpoint, dramatically increases the number of possible paths that can triggered for multivariant executions.

	Original binary		Multivariant WebAssembly binary			
Endpoint	#Functions	#Paths	#Non diversified	#Dispatchers	#Variants	#Paths
libsodium						
encrypt	23	4	18	5	56	325
decrypt	20	3	16	5	49	84
random	8	2	6	2	238	12864
invert	8	2	6	2	125	2784
bin2base64	3	2	1	2	47	172
qrcode-rust						
qr_str	982	$688{*}10^6$	965	17	2092	$97{*}10^{12}$
qr_image	858	$1.4{*}10^6$	843	15	2063	$3{*}10^9$

TABLE 2: Static diversity generated by MEWE, measured on the static call graphs of the WebAssembly binaries, and the preservation of this diversity after translation to machine code. The table is structured as follows: Endpoint name; number of functions and numbers of possible paths in the original WebAssembly binary call graph; number of non diversified functions, number of created dispatchers (one per diversified functions), total number of function variants and number of execution paths in the multivariant WebAssembly binary call graph.

Answer to RQ1

MEWE dramatically increases the number of possible execution paths in the multivariant WebAssembly binary of each endpoint. The large number of possible execution paths, combined with multiple points of random choice in the multivariant call graph thwart the prediction of which path will be taken at runtime.

5.2 RQ2 Results: Intra MTD

To answer RQ2, we execute the multivariant binaries of each endpoint, on the Fastly edge-cloud infrastructure. We execute each endpoint 100 times on each of the 64 Fastly edge nodes. All the executions of a given endpoint are performed with the same input. This allows us to determine if the execution traces are different due to the injected dispatchers and their random behavior. After each execution of an endpoint, we collect the sequence of invoked functions, i.e., the execution trace. Our intuition is that the random dispatchers combined with the function variants embedded in a multivariant binary are very likely to trigger different traces for the same execution, i.e., when an endpoint is executed several times in a row with the same input and on the same edge node. The way both the function variants and the dispatchers contribute to exhibiting different execution traces is illustrated in Figure 4.

Figure 5 shows the ratio of unique traces exhibited by each endpoint, on each of the 64 separate edge nodes. The X corresponds to the edge nodes. The Y axis gives the name of the endpoint. In the plot, for a given (x,y) pair, there is blue point in the Z axis representing Metric 1 over 100 execution traces.

For all edge nodes, the ratio of unique traces is above 0.38. In 6 out of 7 cases, we have observed that the ratio is remarkably high, above 0.9. These results show that MEWE generates multivariant binaries that can randomize execution paths at runtime, in the context of an edge node. The randomization dispatchers, associated to a significant number of function variants greatly reduce the certainty about which computation is performed when running a specific input with a given input value.

Let's illustrate the phenomenon with the endpoint `invert`. The endpoint `invert` receives a vector of integers and returns its inversion. Passing a vector of integers with 100 elements as input, $I = [100, ..., 0]$, results in output $O = [0, ..., 100]$. When the endpoint executes 100 times with the same input on the original binary, we observe 100 times the same execution trace. When the endpoint is executed 100 times with the same input I on the multivariant binary, we observe between 95 and 100 unique execution traces, depending on the edge node. Analyzing the traces we observe that they include only two invocations to a dispatcher, one at the start of the trace and one at the end. The remaining events in the trace are fixed each time the endpoint is executed with the same input I. Thus, the maximum number of possible unique traces is the multiplication of the number of variants for each dispatcher, in this case $29 \times 96 = 2784$. The probability of observing the same trace is $1/2784$.

For multivariant binaries that embed only a few variants, like in the case of the `bin2base64` endpoint, the ratio of unique traces per node is lower than for the other endpoints. With the input we pass to `bin2base64`, the execution trace includes 57 function calls. We have observed that, only one of these calls invokes a dispatcher, which can select among 41 variants. Thus, probability of having the same execution trace twice is $1/41$.

Meanwhile, `qr_str` embeds thousands of variants, and the input we pass triggers the invocation of 3M functions, for which 210666 random choices are taken relying on 17 dispatchers. Consequently, the probability of observing the same trace twice is infinitesimal. Indeed, all the executions of `qr_str` are unique, on each separate edge node. This is shown in Figure 5, where the ratio of unique traces is 1 on all edge nodes.

Answer to RQ2

Repeated executions of a multivariant binary with the same input on an individual edge node exhibits diverse execution traces. MEWE successfully synthesizes multivariant binaries that trigger diverse execution paths at runtime, on individual edge nodes.

5.3 RQ3 Results: Internet MTD

To answer RQ3, we build the union of all the execution traces collected on all edge nodes for a given endpoint. Then, we compute the normalized Shannon Entropy over this set for each endpoint (Metric 2). Our goal is to determine whether the diversity of execution traces we observed on individual

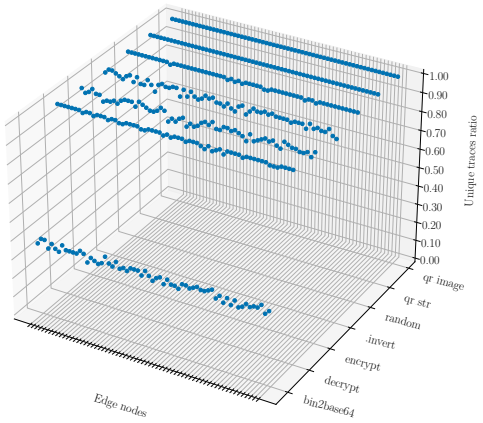

Fig. 5: Ratio of unique execution traces for each endpoint on each edge node. The X axis illustrates the edge nodes. The Y axis annotates the name of the endpoint. In the plot, for a given (x,y) pair, there is blue point representing the Metric 1 value in a set of 100 collected execution traces.

nodes in RQ3, actually generalizes to the whole edge-cloud infrastructure. Depending on many factors, such as the random number generator or a bug in the dispatcher, it could happen that we observe different traces on individual nodes, but that the set of traces is the same on all nodes. With RQ4 we assess the ability of MEWE to exhibit multivariant execution at a global scale.

Table 3 provides the data to answer RQ3. The second column gives the normalized Shannon Entropy value (Metric 2). Columns 3 and 4 give the median and the standard deviation for the length of the execution traces. Columns 5 and 6 give the number of dispatchers that are invoked during the execution of the endpoint (#ED) and the total number of invocations of these endpoints (#Rch). These last two columns indicate to what extent the execution paths are actually randomized at runtime. In the cases of `invert` and `random`, both have the same number of taken random choices. However, the number of variants to chose in `random` are larger, thus, the entropy, is larger than `invert`.

Overall, the normalized Shannon Entropy is above 42%. This is evidence that the multivariant binaries generated by MEWE can indeed exhibit a high degree of execution trace diversity, while keeping the same functionality. The number of randomization points along the execution paths (#Rch) is at the core of these high entropy values. For example, every execution of the `encrypt` endpoint triggers 4M random choices among the different function variants embedded in the multivariant binaries. Such a high degree of randomization is essential to generate very diverse execution traces.

The `bin2base64` endpoint has the lowest level of diversity. As discussed in RQ2, this endpoint is the one that has the least variants and its execution path can be randomized only at one point. The low level of unique traces observed on

Endpoint	Entropy	MTL	σ	#ED	#RCh
libsodium					
encrypt	0.87	816	0	5	4M
decrypt	0.96	440	0	5	2M
random	0.98	15	5	2	12800
invert	0.87	7343	0	2	12800
bin2base64	0.42	57	0	1	6400
qrcode-rust					
qr_str	1.00	3045193	0	17	1348M
qr_image	1.00	3015450	0	15	1345M

TABLE 3: Execution trace diversity over the Fastly edge-cloud computing platform. The table is formed of 6 columns: the name of the endpoint, the normalized Shannon Entropy value (Metric 2), the median size of the execution traces (MTL), the standard deviation for the trace lengths the number of executed dispatchers (#ED) and the number of total random choices taken during all the 6400 executions (#RCh).

individual nodes is reflected at the system wide scale with a globally low entropy.

For both `qr_str` and `qr_image` the entropy value is 1.0. This means that all the traces that we observe for all the executions of these endpoints are different from each other. In other words, someone who runs these services over and over with the same input cannot know exactly what code will be executed in the next execution. These very high entropy values are made possible by the millions of random choices that are made along the execution paths of these endpoints.

While there is a high degree of diversity among the traces exhibited by each endpoint, they all have the same length, except in the case of `random`. This means that the entropy is a direct consequence of the invocations of the dispatchers. In the case of `random`, it naturally has a non-deterministic behavior. Meanwhile, we observe several calls to dispatchers in during the execution of the multivariant binary, which indicates that MEWE can amplify the natural diversity of traces exhibited by `random`. For each endpoint, we managed to trigger all dispatchers during its execution. There is a correlation between the entropy and the number of random choices (Column #RChs) taken during the execution of the endpoints. For a high number of dispatchers, and therefore random choices, the entropy is large, like the cases of `qr_str` and `qr_image` show. The contrary happens to `bin2base64` where its multivariant binary contains only one dispatcher.

Answer to RQ3

At the internet scale of the Edge platform, the multivariant binaries synthesized by MEWE exhibit a massive diversity of execution traces, while still providing the original service. It is virtually impossible for an attacker to predict which is taken for a given query.

5.4 RQ4 Results: Timing side-channels

For each endpoint used in RQ1, we compare the execution time distributions for the original binary and the multivariant binary. All distributions are measured on 100k executions. In Table 4, we show the execution time for the original endpoints and their corresponding multivariant. The table is structured in two sections. The first section shows the endpoint name, the

	Original bin.		Multivariant Wasm	
Endpoint	Median (μs)	σ	Median (μs)	σ
libsodium				
encrypt	7	5	217	43
decrypt	13	6	225	47
random	16	7	232	53
invert	119	34	341	65
bin2base64	10	5	215	35
qrcode-rust				
qr_str	3,117	418	492,606	36,864
qr_image	3,091	412	512,669	41,718

TABLE 4: Execution time distributions for 100k executions, for the original endpoints and their corresponding multivariants. The table is structured in two sections. The first section shows the endpoint name, the median execution time and its standard deviation for the original endpoint. The second section shows the median execution time and its standard deviation for the multivariant WebAssembly binary.

median and standard deviation of the original endpoint. The second section shows the median and the standard deviation for the execution time of the corresponding multivariant binary.

We also observe that the distributions for multivariant binaries have a higher standard deviation of execution time. A statistical comparison between the execution time distributions confirms the significance of this difference (P-value = 0.05 with a Mann-Withney U test). This hints at the fact that the execution time for multivariant binaries is more unpredictable than the time to execute the original binary.

In Figure 6, each subplot represents the distribution for a single endpoint, with the colors blue and green representing the original and multivariant binary respectively. These plots reveal that the execution times are indeed spread over a larger range of values compared to the original binary. This is evidence that execution time is less predictable for multivariant binaries than for the original ones.

We evaluate to what extent a specific variant can be detected by observing the execution time distribution. This evaluation is based on the measurement with one endpoint. For this, we choose endpoint `bin2base64` because it is the end point that has the least variants and the least dispatchers, which is the most conservative assumption.

We dissect the collected execution times for the `bin2base64` endpoint, grouping them by execution path. In Figure 7, each opaque curve represents a cumulative execution time distribution of a unique execution path out of the 41 observed. We observe that no specific distribution is remarkably different from another one. Thus, no specific variant can be inferred out of the projection of all execution times like the ones presented in Figure 6. Nevertheless, we calculate a Mann-Whitney test for each pair of distributions, 41×41 pairs. For all cases, there is no statistical evidence that the distributions are different, $P > 0.05$.

Recall that the choice of function variant is randomized at each function invocation, and the variants have different execution times as a consequence of the code transformations, i.e., some variants execute more instructions than others. Consequently, attacks relying on measuring precise execution times of a function are made a lot harder to conduct as the distribution for the multivariant binary is different and even more spread than the original one.

We note that the execution times are slower for multivariant binaries. Being under 500 ms in general, this does not represent a threat to the applicability of multivariant execution at the edge. Yet, it calls for future optimization research.

Answer to RQ4

The execution time distributions are significantly different between the original and the multivariant binary. Furthermore, no specific variant can be inferred from execution times gathered from the multivariant binary. MEWE contributes to mitigate potential attacks based on predictable execution times.

6 Related Work

Our work is in the area of software diversification for security, a research field discovered by researchers Forrest [46] and Cohen [47]. We contribute a novel technique for multivariant execution, and discuss related work in Section 2. Here, we position our contribution with respect to previous work on randomization and security for WebAssembly.

6.1 Related Work on Randomization

A randomization technique creates a set of unique executions for the very same program [48]. Seminal works include instruction-set randomization [49], [50] to create a unique mapping between artificial CPU instructions and real ones. This makes it very hard for an attacker ignoring the key to inject executable code. Compiler-based techniques can randomly introduce NOP and padding to statically diversify programs. [51] have explored how to use NOP and it breaks the predictability of program execution, even mitigating certain exploits to an extent.

Chew and Song [52] target operating system randomization. They randomize the interface between the operating system and the user applications: the system call numbers, the library entry points (memory addresses) and the stack placement. All those techniques are dynamic, done at runtime using load-time preprocessing and rewriting. Bathkar et al. [48], [53] have proposed three kinds of randomization transformations: randomizing the base addresses of applications and libraries memory regions, random permutation of the order of variables and routines, and the random introduction of random gaps between objects. Dynamic randomization can address different kinds of problems. In particular, it mitigates a large range of memory error exploits. Recent work in this field include stack layout randomization against data-oriented programming [54] and memory safety violations [55], as well as a technique to reduce the exposure time of persistent memory objects to increase the frequency of address randomization [56].

We contribute to the field of randomization, at two stages. First, we automatically generate variants of a given program, which have different WebAssembly code and still behave the same. Second, we randomly select which variant is executed at runtime, creating a multivariant execution scheme that

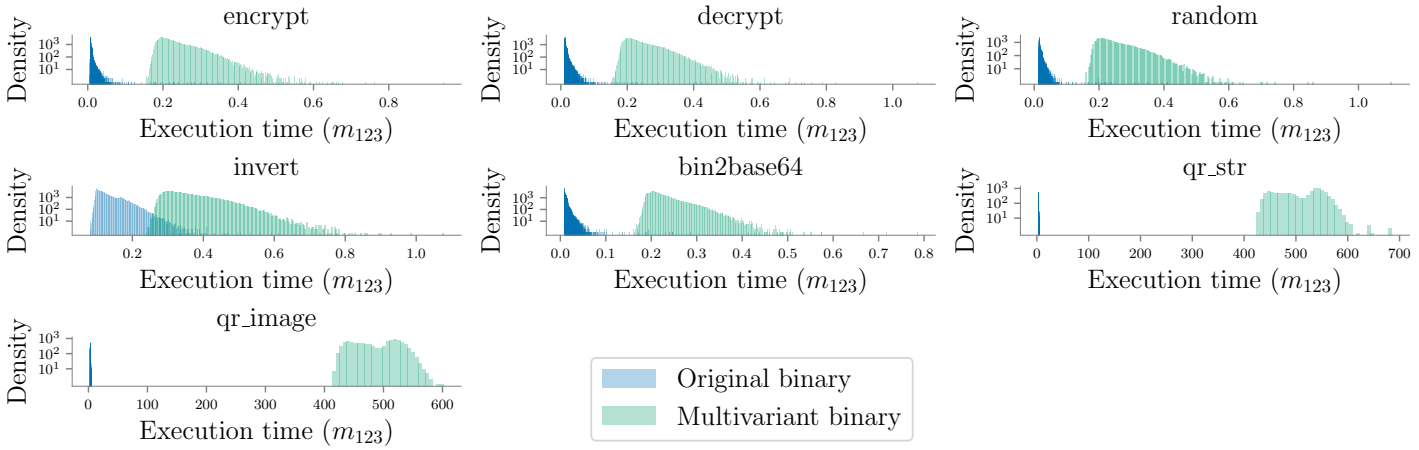

Fig. 6: Execution time distributions. Each subplot represents the distribution for a single endpoint, blue for the original endpoint and green for the multivariant binary. The X axis shows the execution time in milliseconds and the Y axis shows the density distribution in logarithmic scale.

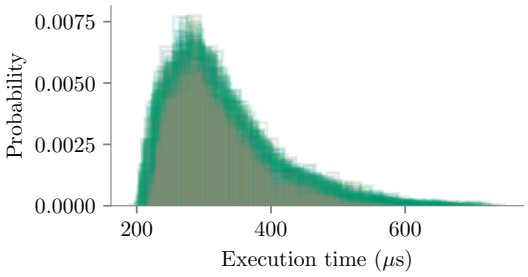

Fig. 7: Execution time distributions for the `bin2base64` endpoint. Each opaque curve represents an execution time distribution of a unique execution path out of the 41 observed.

randomizes the observable execution trace at each run of the program.

Davi et al. proposed Isomeron [57], an approach for execution-path randomization. Isomeron simultaneously loads the original program and a variant. While the program is running, Isomeron continuously flips a coin to decide which copy of the program should be executed next at the level of function calls. With this strategy, a potential attacker cannot predict whether the original or the variant of a program will execute. MEWE proposes two key novel contributions. First, our code diversification step can generate variants of complex control flow structures by inferring constants or loop unrolling. Second, MEWE interconnects hundreds of variants and several randomization dispatchers in a single binary, increasing by orders of magnitude the runtime uncertainty about what code will actually run, compared to the choice among 2 variants proposed by Isomeron.

6.2 Related work on WebAssembly Security

The reference piece about WebAssembly security is by Lehmann et al. [10], which presents three attack primitives. Lehmann et al. have then followed up with a large-scale empirical study of WebAssembly binaries [58]. Narayan et al. [11] remark that the security model of WebAssembly is vulnerable to Spectre attacks. This means that WebAssembly sandboxes may be hijacked and leak memory. They propose to modify the Lucet compiler used by Fastly to incorporate LLVM fence instructions[5] in the machine code generation, trying to avoid speculative execution mistakes. Johnson et al. [43], on the other hand, propose fault isolation for WebAssembly binaries, a technique that can be applied before being deployed to the edge-cloud platforms. Stievenart et al. [59] design a static analysis dedicated to information flow problems. Bian et al. [60] performs runtime monitoring of WebAssembly to detect cryptojacking. The main difference with our work is that our defense mechanism is larger in scope, meant to tackle "yet unknown" vulnerabilities. Notably, MEWE is agnostic from the last-step compilation pass that translates Wasm to machine code, which means that the multivariant binaries can be deployed on any edge-cloud platform that can receive WebAssembly endpoints, regardless of the underlying hardware.

7 Discussion

Specialising, optimizing, improving performance In Section 4 we validated the key features of MEWE: the automatic generation of multivariant binaries, which exhibit random execution paths at runtime. Several aspects of these procedures can be optimized. For example, the generated code can be optimized by inline function variants in the dispatchers. This minimal change will decrease the number of function calls. On the other hand, the number of times a dispatcher is called can be bound. As discussed in RQ3 and RQ4, the dispatchers are massively invoked at runtime, which is great for randomization, but also a challenge with respect to the execution time of the services.

Fuzzing and security The diversification created by MEWE can unleash hidden behaviors in compilers like Lucet. One of the biggest challenges in fuzzing compilers is the ability to reach latter stages in the machine code generation pipeline. By generating several functionally equivalent, and yet different variants, deeper bugs can be discovered. During

5. https://llvm.org/doxygen/classllvm_1_1FenceInst.html

the writing of this work, an error was discovered during the execution of one of the variants provided by MEWE. Fastly acknowledged our work as part of a technical blog post [6] that describes the bug and the patch.

The variants generated by MEWE in WebAssembly are preserved by the Lucet compiler. We checked for code diversity preservation after compilation. In this work, diversity is introduced through transformation on WebAssembly code, which is then compiled by the Lucet compiler. Compilation might perform some normalization and optimization passes when translating from WebAssembly to machine code. Thus, some variants synthesized by MEWE might not be preserved, i.e., Lucet could generate the same machine code for two WebAssembly variants. To assess this potential effect, we compare the level of code diversity among the WebAssembly variants and among the machine code variants produced by Lucet. This experiment reveals that the translation to machine code preserves a high ratio of function variants, i.e., approx 96% of the generated variants are preserved. This result also indicates that the machine code variants preserve the potential for large numbers of possible execution paths.

8 Conclusion

In this work we propose a novel technique to automatically synthesize multivariant binaries to be deployed on edge computing platforms. Our tool, MEWE, operates on a single service implemented as a WebAssembly binary. It automatically generates functionally equivalent variants for each function that implements the service, and combines all the variants in a single WebAssembly binary, which exact execution path is randomized at runtime. Our evaluation with 7 real-world cryptography and QR encoding services shows that MEWE can generate hundreds of function variants and combine them into binaries that include from thousands to millions of possible execution paths. The deployment and execution of the multivariant binaries on the Fastly cloud platform showed that they actually exhibit a very high diversity of execution at runtime, in single edge nodes, as well as Internet scale.

Future work with MEWE will address the trade-off between a large space for execution path randomization and the computation cost of large-scale runtime randomization. In addition, the synthesis of a large pool of variants supports the exploration of the concurrent execution of multiple variants to detect misbehaviors in services deployed at the edge.

References

[1] S. Choy, B. Wong, G. Simon, and C. Rosenberg, "A hybrid edge-cloud architecture for reducing on-demand gaming latency," *Multimedia systems*, vol. 20, no. 5, pp. 503–519, 2014.

[2] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration," *IEEE Comm. Surveys & Tutorials*, vol. 19, no. 3, 2017.

[3] "The New York Times on failure, risk, and prepping for the 2016 US presidential election – Fastly," 2021. [Online]. Available: https://www.fastly.com/blog/new-york-times-on-failure-risk-and-prepping-2016-us-presidential-election

[4] K. Varda, "Webassembly on cloudflare workers," Tech. Rep., Jan. 2018. [Online]. Available: https://blog.cloudflare.com/webassembly-on-cloudflare-workers/

6. The link to the post will be provided with the deanonymized version of this paper.

[5] P. Hickey, "Announcing lucet: Fastly's native webassembly compiler and runtime," Tech. Rep., Mar. 2018. [Online]. Available: https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime

[6] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 185–200.

[7] D. Bryant, "Webassembly outside the browser: A new foundation for pervasive computing," in *Proc. of ICWE 2020*, 2020, pp. 9–12.

[8] P. Mendki, "Evaluating webassembly enabled serverless approach for edge computing," in *2020 IEEE Cloud Summit*, 2020, pp. 161–166.

[9] N. Silvanovich, "The problems and promise of webassembly," Tech. Rep., Mar. 2018. [Online]. Available: https://googleprojectzero.blogspot.com/2018/08/the-problems-and-promise-of-webassembly.html

[10] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of webassembly," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020.

[11] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen *et al.*, "Swivel: Hardening webassembly against spectre," in *USENIX Security Symposium*, 2021.

[12] "Global CDN Disruption," 2021. [Online]. Available: https://status.fastly.com/incidents/vpk0ssybt3bj

[13] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: a secretless framework for security through diversity," in *Proc. of USENIX Security Symposium*, ser. USENIX-SS'06, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267336.1267344

[14] J. Cabrera-Arteaga, O. F. Malivitsis, O. Vera-Pérez, B. Baudry, and M. Monperrus, "Crow: Code diversification for webassembly," in *MADWeb, NDSS 2021*, 2021.

[15] M. Jacobsson and J. Wåhslén, "Virtual machine execution for wearables based on webassembly," in *EAI International Conference on Body Area Networks*. Springer, Cham, 2018, pp. 381–389.

[16] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *USENIX Annual Technical Conference*, 2020, pp. 419–433.

[17] " The LLVM Compiler Infrastructure ," 2021. [Online]. Available: https://llvm.org/

[18] "Binaryen," 2021. [Online]. Available: https://github.com/WebAssembly/binaryen

[19] D. Bruschi, L. Cavallaro, and A. Lanzi, "Diversified process replicæ for defeating memory error exploits," in *Proc. of the Int. Performance, Computing, and Communications Conference*, 2007.

[20] B. Salamat, A. Gal, T. Jackson, K. Manivannan, G. Wagner, and M. Franz, "Stopping buffer overflow attacks at run-time: Simultaneous multi-variant program execution on a multicore processor," Technical Report 07-13, School of Information and Computer Sciences, UCIrvine, Tech. Rep., 2007.

[21] T. Jackson, C. Wimmer, and M. Franz, "Multi-variant program execution for vulnerability detection and analysis," in *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, 2010, pp. 1–4.

[22] K. Lu, M. Xu, C. Song, T. Kim, and W. Lee, "Stopping memory disclosures via diversification and replicated execution," *IEEE Transactions on Dependable and Secure Computing*, 2018.

[23] S. Volckaert, B. Coppens, and B. De Sutter, "Cloning your gadgets: Complete rop attack immunity with multi-variant execution," *IEEE Transactions on Dependable and Secure Computing*, vol. 13, no. 4, 2015.

[24] B. Salamat, T. Jackson, G. Wagner, C. Wimmer, and M. Franz, "Runtime defense against code injection attacks using replicated execution," *IEEE Trans. Dependable Secur. Comput.*, vol. 8, no. 4, pp. 588–601, 2011. [Online]. Available: https://doi.org/10.1109/TDSC.2011.18

[25] S. Österlund, K. Koning, P. Olivier, A. Barbalace, H. Bos, and C. Giuffrida, "kmvx: Detecting kernel information leaks with multi-variant execution," in *ASPLOS*, 2019.

[26] B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space," in *Proceedings of the 4th ACM European conference on Computer systems*, 2009, pp. 33–46.

[27] K. Koning, H. Bos, and C. Giuffrida, "Secure and efficient multi-variant execution using hardware-assisted process virtualization,"

in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2016, pp. 431–442.
[28] N. Belleville, D. Couroussé, K. Heydemann, and H.-P. Charles, "Automated software protection for the masses against side-channel attacks," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 4, nov 2018. [Online]. Available: https://doi.org/10.1145/3281662
[29] M. Maurer and D. Brumley, "Tachyon: Tandem execution for efficient live patch testing," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 617–630.
[30] D. Kim, Y. Kwon, W. N. Sumner, X. Zhang, and D. Xu, "Dual execution for on the fly fine grained execution comparison," *SIGPLAN Not.*, 2015.
[31] A. Voulimeneas, D. Song, P. Larsen, M. Franz, and S. Volckaert, "dmvx: Secure and efficient multi-variant execution in a distributed setting," in *Proceedings of the 14th European Workshop on Systems Security*, 2021, pp. 41–47.
[32] D. J. Bernstein, "Cache-timing attacks on aes," 2005.
[33] O. Acıiçmez, W. Schindler, and Ç. K. Koç, "Cache based remote timing attack on the aes," in *Cryptographers' track at the RSA conference*. Springer, 2007, pp. 271–286.
[34] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 431–446.
[35] A. J. O'Donnell and H. Sethu, "On achieving software diversity for improved network security using distributed coloring algorithms," in *Proceedings of the 11th ACM conference on Computer and communications security*, 2004, pp. 121–131.
[36] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity," in *NDSS*, 2015, pp. 8–11.
[37] T. Brennan, N. Rosner, and T. Bultan, "Jit leaks: inducing timing side channels through just-in-time compilation," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1207–1222.
[38] H. Liljestrand, T. Nyman, L. J. Gunn, J.-E. Ekberg, and N. Asokan, "Pacstack: an authenticated call stack," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
[39] S. Allier, O. Barais, B. Baudry, J. Bourcier, E. Daubert, F. Fleurey, M. Monperrus, H. Song, and M. Tricoire, "Multitier diversification in web-based software applications," *IEEE Software*, vol. 32, no. 1, pp. 83–90, 2015. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01089268/file/final-multi-tier.pdf
[40] B. Coppens, B. De Sutter, and J. Maebe, "Feedback-driven binary code diversification," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–26, 2013.
[41] J. Lettner, D. Song, T. Park, P. Larsen, S. Volckaert, and M. Franz, "Partisan: fast and flexible sanitization via run-time partitioning," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2018, pp. 403–422.
[42] B. G. Ryder, "Constructing the call graph of a program," *IEEE Transactions on Software Engineering*, no. 3, pp. 216–226, 1979.
[43] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Sfi safety for native-compiled wasm," *NDSS. Internet Society*, 2021.
[44] "Webassembly system interface," 2021. [Online]. Available: https://github.com/WebAssembly/WASI
[45] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Ann. Math. Statist.*, vol. 18, no. 1, pp. 50–60, 03 1947.
[46] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems*. IEEE, 1997, pp. 67–72.
[47] F. B. Cohen, "Operating system protection through program evolution." *Computers & Security*, vol. 12, no. 6, pp. 565–584, 1993.
[48] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: an efficient approach to combat a board range of memory error exploits," in *Proceedings of the USENIX Security Symposium*, 2003.
[49] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proc. of CCS*, 2003, pp. 272–280.
[50] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proc. CCS*, 2003, pp. 281–289.
[52] M. Chew and D. Song, "Mitigating buffer overflows by operating system randomization," Carnegie Mellon University, Tech. Rep. CS-02-197, 2002.
[51] T. Jackson, "On the design, implications, and effects of implementing software diversity for security," Ph.D. dissertation, University of California, Irvine, 2012.
[53] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *Proceedings of the USENIX Security Symposium*, 2005, pp. 271–286.
[54] M. T. Aga and T. Austin, "Smokestack: thwarting dop attacks with runtime stack layout randomization," in *Proc. of CGO*, 2019, pp. 26–36. [Online]. Available: https://drive.google.com/file/d/12TvsrgL8Wt6IMfe6ASUp8y69L-bCVao0/view
[55] S. Lee, H. Kang, J. Jang, and B. B. Kang, "Savior: Thwarting stack-based memory safety violations by randomizing stack layout," *IEEE Transactions on Dependable and Secure Computing*, 2021. [Online]. Available: https://ieeexplore.ieee.org/iel7/8858/4358699/09369900.pdf
[56] Y. Xu, Y. Solihin, and X. Shen, "Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization," in *Proc. of ASPLOS*, 2020, pp. 987–1000. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3373376.3378492
[57] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," in *NDSS*, 2015.
[58] A. Hilbig, D. Lehmann, and M. Pradel, "An empirical study of real-world webassembly binaries: Security, languages, use cases," in *Proceedings of the Web Conference 2021*, 2021, pp. 2696–2708.
[59] Q. Stiévenart and C. De Roover, "Compositional information flow analysis for webassembly programs," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2020, pp. 13–24.
[60] W. Bian, W. Meng, and M. Zhang, "Minethrottle: Defending against wasm in-browser cryptojacking," in *Proceedings of The Web Conference 2020*, 2020, pp. 3112–3118.

Javier Cabrera Arteaga is currently a PhD student in the Software and Computer Systems department, KTH Royal Institute of Technology. His current research interests include software diversification, automated software testing and compilers.

Pierre Laperdrix is currently a research scientist for CNRS in the Spirals team in the CRIStAL laboratory in Lille, France. He obtained his PhD working on browser fingerprinting at INRIA Rennes. His research interests span several areas of security and privacy with a strong focus on the web. One of his main goal is to understand what is happening on the web to ultimately design countermeasures to better protect users online.

Martin Monperrus is Professor of Software Technology at KTH Royal Institute of Technology. He received a Ph.D. from the University of Rennes, and a Master's degree from Compiègne University of Technology. His research lies in the field of software engineering with a current focus on automatic program repair, program hardening and chaos engineering.

Benoit Baudry is a Professor in Software Technology at the KTH Royal Institute of Technology, and the director of the CASTOR software research center. He received his PhD in 2003 from the University of Rennes, France and was a research scientist at INRIA from 2004 to 2017. His research interests include automated software engineering, software diversity and software testing. In 2017, he received a WASP endowed chair to support his research at KTH, and in 2018 he received the CNIL-INRIA "Privacy Protection" Award for his work on browser fingerprinting.