Engineering Software Diversity: a Model-Based Approach to Systematically Diversify Communications*

Brice Morin, Jakob Høgenes and Hui Song
SINTEF Digital
Oslo, Norway
first.last@sintef.no

Nicolas Harrand and Benoit Baudry
KTH
Stockholm, Sweden
last@kth.se

ABSTRACT

Automated diversity is a promising mean of increasing the security of software systems. However, current automated diversity techniques operate at the bottom of the software stack (operating system and compiler), yielding a limited amount of diversity. We present a novel Model-Driven Engineering approach to the diversification of communicating systems, building on abstraction, model transformations and code generation. This approach generates significant amounts of diversity with a low overhead, and addresses a large number of communicating systems, including small communicating devices.

ACM Reference Format:
Brice Morin, Jakob Høgenes and Hui Song and Nicolas Harrand and Benoit Baudry. 2018. Engineering Software Diversity: a Model-Based Approach to Systematically Diversify Communications. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18), October 14–19, 2018, Copenhagen, Denmark*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3239372.3239393

1 INTRODUCTION

In "mass-produced" software, *e.g.* the firmware of a connected device or a mobile/web application, diversity is automatically introduced either *a*) in a generic way, typically at the OS level, oblivious from the actual logic and semantics of the software, or *b*) in some very specific places, typically low-level libraries reused across applications, in order to improve security. This left most of the actual business logic unchanged, unaffected by the diversity [1].

A more holistic approach to diversity is challenging [2, 3]. Consider a typical client-server application, where multiple clients interact with a server, and where each client has a different implementation, and a different way of communicating with the server. This would significantly reduce the observability, learnability and predictability of the overall system [4, 5] *i.e.*, reduce the capability to generalize from one example. This would make large-scale exploits, such as MIRAI [6] or PETYA [7], a time-consuming and costly endeavor for hackers. Yet, the engineering, e.g., the production, maintenance and integration, of such levels of diversity raises several challenges. How to ensure that each implementation still behaves as specified? How to ensure that each client still can communicate with the server, without information loss or distortion? How to ensure that different clients are fundamentally different, and not merely cosmetically different? How to keep the development and operation costs of a diversified system significantly lower than the cost of mitigating large scale attacks?

In this work we introduce an original, model-driven approach for engineering software diversity. We focus on the synthesis of software diversity into the communications between different machines or processes, for example between a device and a gateway, or a web/mobile app and a server. This specific scope is motivated by the increasingly strategic role communications play in an always more connected world. For example, a number of attacks have emerged to by-pass security, such as the KRACK attack on WPA2 (securing WiFi) [8], calling for new counter-measures to reduce the certainty that an attacker can have about interactions between communicating entities.

Our approach to the diversification of communications follows the principles of Model-Driven Engineering (MDE):

(1) we abstract communications into protocol models, composed of *a*) a structural view describing the messages to be exchanged by two (or more) participants, and *b*) a behavioral view describing how those messages are exchanged between the participants, including sequencing and timing.
(2) we apply and combine together a number of atomic model transformations onto this protocol model, aiming at deeply changing the way the protocol operates, while still preserving its semantics. The precise description of these atomic transformations and the detailed evaluation of their combined impact is the main contribution of this paper.
(3) we automatically generate code implementing each of the diversified protocol models

This MDE approach to the diversification of communications addresses some of the key challenges to engineer software diversity. First, it automatically generates fundamentally different communications, from a single specification. Our experiments with 200 protocols, diversified from a single base protocol confirm this huge diversity. This can significantly reduce the predictability and harmful exploitation of communication protocols. Second, it yields fully operational code, which can run on a wide number of platforms and communicate through a large number of transport protocols. Our empirical assessment of the approach indicates that it implies a reasonable overhead in terms of execution time, memory consumption and bandwidth. On powerful nodes able to run JavaScript or Go,

*The research leading to these results has received funding from the European Commission's H2020 programme under grant agreement no 780351 (ENACT), and from the Norwegian Research Council IKT-PLUSS programme under grant agreement no 270933 (DiversIoT).

MODELS '18, October 14–19, 2018, Copenhagen, Denmark
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-4949-9/18/10…$15.00
https://doi.org/10.1145/3239372.3239393

this overhead will be imperceptible in practice: less than +100 µs of processing time per message, +3 bytes per message exchanged on the network, and less than +300 kB of RAM to handle a diversified protocol. On Arduino, a resource-constrained microcontroller (2kB RAM, 32kB Flash, 16MHz MCU) representative of small Internet-of-Things devices, this overhead is +1ms of processing time per message, +3 bytes also per message exchanged on the network and +150 bytes of RAM to handle a diversified protocol.

The remainder of this paper is organized as follows. Section 2 presents the overall model-driven process to diversity communications. Section 3 details the model-transformations we use to implement the diversification. Section 4 evaluates our approach by assessing the diversity and evaluating the overhead induced by our approach. Section 5 discusses related work. Section 6 concludes and presents future work.

2 MODEL-DRIVEN PROCESS TO DIVERSIFY COMMUNICATIONS

This section gives an overview of the overall model-driven process to diversify communications. First, we present how protocol models are specified (Section 2.1). Next, we introduce the diversification process itself (Section 2.2), based on the atomic model transformations further detailed in Section 3. Finally, we present how fully operational code is generated from those diversified models (Section 2.3).

2.1 Specifying Protocol Models

We rely on ThingML [9] for the specification of protocol models. ThingML is a modeling language initially developed to model the reactive behavior of a set of “Internet-of-Things” devices communicating through asynchronous message passing. Through a decade of research and development, ThingML has evolved to a more general modeling language, able to represent the behavior of most distributed systems communicating through asynchronous message passing. Fundamentally, ThingML is based on a sub-set of the UML (statecharts and components), with a textual syntax bridging the gap with formalisms developers are used to *i.e.*, code. By default, ThingML comes with a set of compilers targeting C (for microcontrollers and Linux), Java, JavaScript (Node.JS and Browser) and Go, and a set of plugins to allow components to communicate through a number of network protocols such as MQTT, WebSocket, and so on, which we use in Section 2.3.

We model communication protocols as a set of communicating state-machines, encapsulated into components. A protocol involves two roles: 1) a client *e.g.*, a device, a web-browser or a mobile app, and 2) a server *e.g.*, a gateway or a cloud back-end. In the remainder of this paper, we assume that communications happen between one server and a number of clients.

First, the clients and the server need to agree on a common API. Since communication is typically asynchronous in a distributed system, the common API is specified as a set of messages. Listing 1 illustrates an example for API specification.

Listing 1: A simple API

```
thing fragment Msgs {
 message m1(a:Byte,b:Byte,c:Integer,
            d:Byte,e:Byte)
 message m2(a:Byte,b:Byte,c:Byte)
 message m3(a:Byte)
}
```

A thing fragment is a form of interface, defining a set of structural features, here three messages, that can later on be included into other things. This interface defines three messages m1, m3 and m3. Each message encapsulates data into a set of parameters, and each parameter is typed.

Next, this API is imported by the client component and the server component, and the messages are organized into ports, as shown in the script below:

Listing 2: The client component

```
thing Client includes Msgs {
 required port app {
  sends m1, m2
  receives m3
 }
}
```

The client component, or thing, sends messages m1 and m2, and receives message m3. Symmetrically, the server receives m1 and m2, and sends m3. The overall structure for this simple protocol is shown in Figure 1.

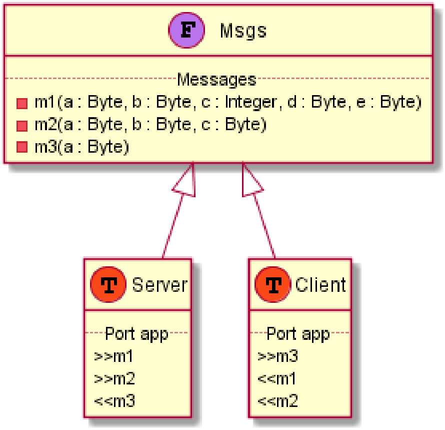
F
Msgs
Messages
m1(a : Byte, b : Byte, c : Integer, d : Byte, e : Byte)
m2(a : Byte, b : Byte, c : Byte)
m3(a : Byte)
T
Server
Port app
>>m1
>>m2
<<m3
T
Client
Port app
>>m3
<<m1
<<m2

Figure 1: API for the simple protocol

Finally, the behavior of the client and the server needs to be implemented. Though ThingML allows for the complete implementation of components, in this paper we are only concerned about implementing the logic describing valid interactions, *e.g.* sequences and timing related to the emission or reception of messages. All the behavior beyond this logic is not relevant for this paper.

Figure 2 shows the implementation of the protocol for the client. In this example, the client will be identified by a random number, stored in variable _a, which is initialized on startup. In the RUN state, the client will 1) send messages m1 and m2, containing the identifier of the client, together with other data, and 2) wait for a response from the server. The server replies with a message m3. If the parameter a of this message m3 is different from the identifier of the client, the client goes in the ERROR state and terminates. If the parameter is valid, the client re-enters the RUN state. After 100 successful interactions, the client will terminate.

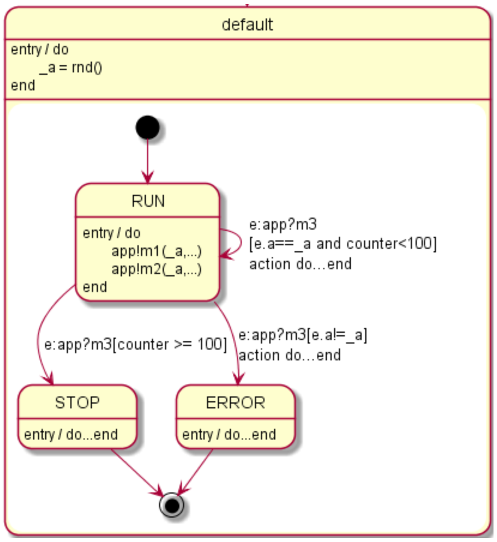

Figure 2: Behavior of the client, exported from ThingML

In a rather symmetrical way, the server will 1) wait for a message m1, then 2) wait for a message m2, and finally 3) if the client identifier is correct, sends a message m3 back to the client with this identifier.

2.2 Diversifying Protocol Models through Model Transformations

The model-based diversification process itself is depicted in Figure 3. The diversifier is a *1-to-n*, endogenous transformation. It takes a ThingML protocol model as input and generates *n* ThingML protocol models as output. A key benefit of this design choice is that the whole ThingML tool-chain, and in particular the compilers (Section 2.3), can be reused as-is on those diversified protocols.

The transformation is configured according to the following parameters:

(1) *seed*: the seed for the random number generator used internally in the diversifier. Given one seed and one input protocol, the diversifier produces repeatable outputs.

(2) *mode*: all transformations introduce static diversity. In addition, some transformation (see Sections 3.5 and 3.6) can also introduce dynamic diversity.

  (a) *static diversity*: in this mode, the diversifier generates different implementations for the input protocol. However, all the diversity is fixed by the diversifier itself *i.e.*, the protocol will not evolve over time.

  (b) *dynamic diversity*: in this mode, the diversifier still generates different implementations. In addition, some diversity still remains open at runtime, so as to allow the protocol to change over time (within some boundaries) and act as a moving target defense [10].

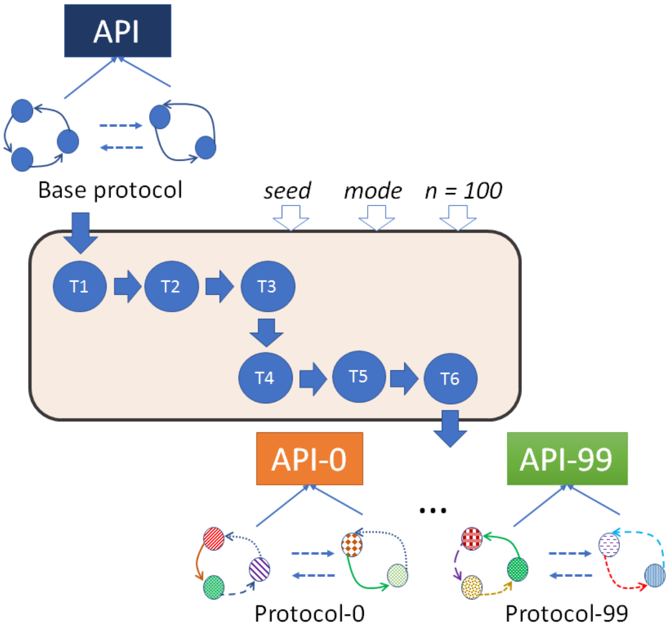

Figure 3: Overview of the model-based diversification process for communication protocols

(3) *n*: the number of diversified protocols to be generated. In Section 4, we generate 100 protocols in each mode.

Figure 4 shows one diversified protocol for the client, which has been generated from the inputs described by Figure 1 and Figure 2. Note that for this diversified client, and every other diversified client, a corresponding protocol is also generated for the server.

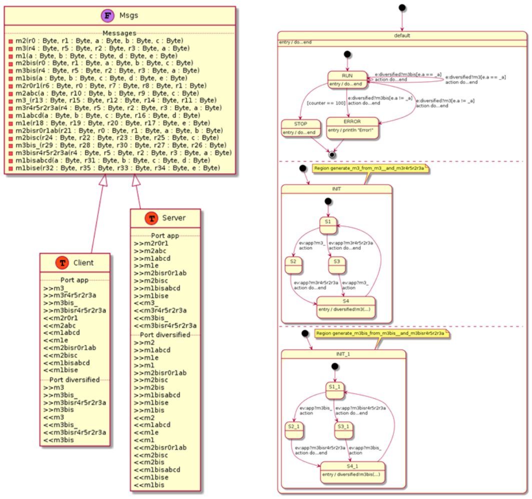
F Msgs
Messages
m2(r0 : Byte, r1 : Byte, a : Byte, b : Byte, c : Byte)
m3(r4 : Byte, r5 : Byte, r2 : Byte, r3 : Byte, a : Byte)
m1(a : Byte, b : Byte, c : Byte, d : Byte, e : Byte)
m2bis(r0 : Byte, r1 : Byte, a : Byte, b : Byte, c : Byte)
m3bis(r4 : Byte, r5 : Byte, r2 : Byte, r3 : Byte, a : Byte)
m1bis(a : Byte, b : Byte, c : Byte, d : Byte, e : Byte)
m2r0r1(r6 : Byte, r0 : Byte, r7 : Byte, r8 : Byte, r1 : Byte)
m2abc(a : Byte, r10 : Byte, b : Byte, r9 : Byte, c : Byte)
m3_(r13 : Byte, r15 : Byte, r12 : Byte, r14 : Byte, r11 : Byte)
m3r4r5r2r3a(r4 : Byte, r5 : Byte, r2 : Byte, r3 : Byte, a : Byte)
m1abcd(a : Byte, b : Byte, c : Byte, r16 : Byte, d : Byte)
m1e(r18 : Byte, r19 : Byte, r20 : Byte, r17 : Byte, e : Byte)
m2bisr0r1ab(r21 : Byte, r0 : Byte, r1 : Byte, a : Byte, b : Byte)
m2bisc(r24 : Byte, r22 : Byte, r23 : Byte, r25 : Byte, c : Byte)
m3bis_(r29 : Byte, r28 : Byte, r30 : Byte, r27 : Byte, r26 : Byte)
m3bisr4r5r2r3a(r4 : Byte, r5 : Byte, r2 : Byte, r3 : Byte, a : Byte)
m1bisabcd(a : Byte, r31 : Byte, b : Byte, c : Byte, d : Byte)
m1bise(r32 : Byte, r35 : Byte, r33 : Byte, r34 : Byte, e : Byte)
T Client
Port app
>>m3_
>>m3r4r5r2r3a
>>m3bis_
>>m3bisr4r5r2r3a
<<m2r0r1
<<m2abc
<<m1abcd
<<m1e
<<m2bisr0r1ab
<<m2bisc
<<m1bisabcd
<<m1bise
Port diversified
>>m3
>>m3bis_
>>m3bisr4r5r2r3a
>>m3bis
<<m3
<<m3bis_
<<m3bisr4r5r2r3a
<<m3bis
T Server
Port app
>>m2r0r1
>>m2abc
>>m1abcd
>>m1e
>>m2bisr0r1ab
>>m2bisc
>>m1bisabcd
>>m1bise
<<m3_
<<m3r4r5r2r3a
<<m3bis_
<<m3bisr4r5r2r3a
Port diversified
>>m2
>>m1abcd
>>m1e
>>m1
>>m2bisr0r1ab
>>m2bisc
>>m2bis
>>m1bisabcd
>>m1bise
>>m1bis
<<m2
<<m1abcd
<<m1e
<<m1
<<m2bisr0r1ab
<<m2bisc
<<m2bis
<<m1bisabcd
<<m1bise
<<m1bis
default
entry / do...end
RUN
entry / do...end
e:diversified?m3bis[e.a == _a]
action do...end
e:diversified?m3[e.a == _a]
action do...end
[counter == 100]
e:diversified?m3bis[e.a != _a]
action do...end
e:diversified?m3[e.a != _a]
action do...end
STOP
entry / do...end
ERROR
entry / println "Error!"
Region generate_m3_from_m3__and_m3r4r5r2r3a
INIT
S1
ev:app?m3_
action
ev:app?m3r4r5r2r3a
action do...end
S2
S3
ev:app?m3r4r5r2r3a
action do...end
ev:app?m3_
action
S4
entry / diversified!m3(...)
Region generate_m3bis_from_m3bis__and_m3bisr4r5r2r3a
INIT_1
S1_1
ev:app?m3bis_
action
ev:app?m3bisr4r5r2r3a
action do...end
S2_1
S3_1
ev:app?m3bisr4r5r2r3a
action do...end
ev:app?m3bis_
action
S4_1
entry / diversified!m3bis(...)

Figure 4: A diversified client, exported from ThingML

A detailed discussion about this particular diversified protocol is beyond the scope of this paper as it represents a very small sample of the quasi-infinite number of protocols that can be generated by our approach. We will detail in Section 3 the six atomic model transformations contributing to producing diversified protocols.

2.3 Generating Fully Operational Code from (Diversified) Protocol Models

The automatic generation of both client and server side code is an essential feature of our MDE process for the diversification of communication.

Here, we rely on the compilers available as part of the ThingML Framework. The diversified protocol models "plain old ThingML specifications", and those compilers[1] can be reused out-of-the-box. This way we can generate C (for resource-constrained microcontrollers or Linux), Java, JavaScript (for the browser or Node.JS) or Go code for the client and server sides.

Given a communication model, different compilers generate different versions of the machine code, where the control flow and data flow to manage incoming and outgoing messages is likely to use different low-level instructions, storing intermediate data differently in RAM and in registers. A discussion on this computing diversity is beyond the scope of this paper, and pending further work. In this paper, we focus on the communications between components, and consider each component as a black-box.

ThingML supports communication [11] by generating the 'glue' between components:

(1) marshalling/unmarshalling logic, to serialize and parse message in a string or binary format. In this work we use the binary format, which is similar to Google's Protocol Buffer [12]
(2) transport logic, to transfer message between two local processes, or between two remote processes over the network. For local communications, ThingML relies on in-memory queues where processes read and write. For networked communications, ThingML generates wrappers around components, which delegate this logic to existing libraries for HTTP, WebSocket, MQTT, Serial, and so on.

Figure 5 shows how messages are serialized. The first byte of any message is an identification code, necessary for the receiving side to parse the message. The following bytes encode the parameters, which can have different size depending on their types. If parameter a is an identifier for the client, it represents an essential piece of information for attackers: knowing this, they can focus on the other parameters, see how they evolve over time, how they correlate, etc. This is exactly the learnability that we aim at breaking, or making more difficult, using the transformations described in Section 3.

Once the code is generated, it must be deployed and executed. One potential issue with our approach is that, for a high number N of clients, if we assume each client is provided with a unique protocol, the server needs to run N versions of the server-side protocol. This would be a major issue, and even a show stopper, in a static monolithic system, where those N versions would need to be built in this monolith and kept up and running at any time. We see this as a purely theoretical issue, as companies still operating a static monolith in 2018 most likely do not deal with a very high number of users. In contrary, leading companies such as Netflix, Google, Amazon and so on, are already running a large number of instances on the server side. More recent, dynamic and scalable architectures, such as serverless architectures (a.k.a Function-as-a-Service)[2] are perfectly compatible with our approach, able to run those server-side protocols only when needed, at a low operational cost.

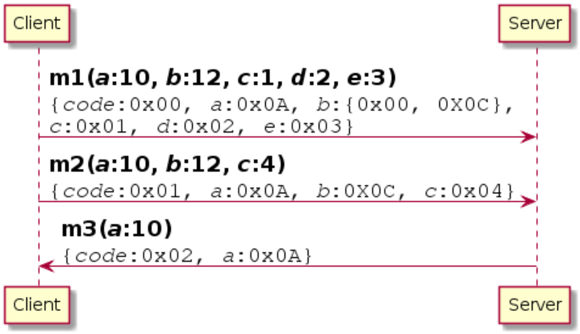
Client
Server
m1(a:10, b:12, c:1, d:2, e:3)
{code:0x00, a:0x0A, b:{0x00, 0X0C}, c:0x01, d:0x02, e:0x03}
m2(a:10, b:12, c:4)
{code:0x01, a:0x0A, b:0X0C, c:0x04}
m3(a:10)
{code:0x02, a:0x0A}
Client
Server

Figure 5: Serialization of the protocol

2.4 Implementation

ThingML and our communications diversification tool are publicly available on GitHub under the open-source Apache 2.0 License:

- **ThingML**[3]: Our approach builds on ThingML, a modeling language and code generation framework for reactive systems, developed over the past 10 years.
- **thingml-diversifier**[4]: Our approach to diversity communications.
  - *src/main/java* contains the implementation of the diversifier *i.e.*, the model transformations described in Section 3.
  - *src/main/python* contains the evaluation procedure used in Section 4.

3 MODEL-BASED DIVERSIFICATION STRATEGIES FOR COMMUNICATIONS

Diversifications are introduced into communications through a set of model transformations. Fundamentally, the goal of these transformations is to break sequences and patterns for individual client/server pairs, and to reduce the similarities between different client/server pairs.

3.1 Re-ordering messages

This transformation re-orders the definitions of messages that a ThingML component can send and receive. Remember that each message is associated with an identifier, serialized as `bytes[0]` in the payload to be sent on the network, so that it can be parsed by the receiving side. Those identifiers are generated by ThingML as a sequence following the order of message definitions. Re-ordering messages will thus prevent the same message to always be affected with the same code for different clients, and will contribute to

[1] strictly speaking, model-to-text transformations generating source code, which then can be interpreted or compiled to machine code by other compilers (gcc, javac, etc).

[2] See for example Amazon's AWS Lambda (https://aws.amazon.com/lambda/) or Microsoft's Azure Functions (https://azure.microsoft.com/en-us/services/functions/)

[3] https://github.com/TelluIoT/ThingML

[4] https://github.com/SINTEF-9012/thingml-diversifier

increase the diversity and reduce the learnability of the code generated from ThingML.

This simple transformation is basically implemented as:

```
java.util.Collections.shuffle(myCpt.getMessages())
```

Figure 6 shows the impact of this transformation on the payloads to be exchanged on the network. Here, three clients (12, 36 and 99) send a message `m2` to the server. The first parameter `a` contains the identifier of the clients (`0x0C`, `0x23` and `0x63`, hexadecimal notation), while the other parameters are set to zero.

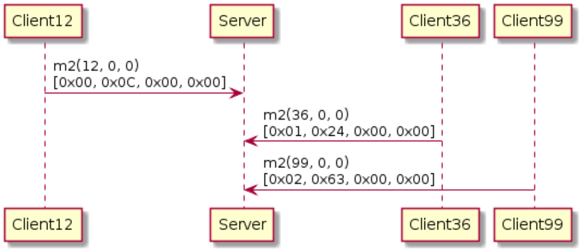

Figure 6: Impact of re-ordering messages on three clients

The first bytes of the payloads, identifying the same message `m2` for different clients, is different: `0x00`, `0x01` and `0x02`.

3.2 Re-ordering parameters

This transformation re-orders the parameters within a given message and is implemented in a similar way as the previous transformation. In addition to re-shuffling the order of the formal parameters defined in the message definitions, the order of the actual parameters when a message is sent should also be updated accordingly. In ThingML, sending a message is a first-class concept in the metamodel. For all instances of `SendAction` referring to this message, the list of their actual parameters is re-shuffled according to the same order.

This is quite similar to refactoring a function expressed in any programming language. Should you update the order of the function's parameters, all calls to this function should also be updated to ensure the consistency of this refactoring at the interface level, avoid type errors and bugs in the refactored program and preserve its initial semantics. This implies side-effects observable at a lower level: when calling the function, the order in which their actual parameters are pushed to the stack and/or the order in which registers are allocated to actual parameters will be different. In the same way, this transformation will impact the order in which the parameter of a message are serialized.

Note that transitions reacting to the original message do not need to be updated even though they might use the values of the parameters. In ThingML, those values will be accessed by referring to the name of the parameter and not by referring to their positions.

Figure 7 shows the impact of this transformation. In particular, it is important to note that the identifier for each client is not anymore at a fixed place in the payload: `bytes[2]` for client 12, `bytes[3]` for client 36 and `bytes[1]` for client 99.

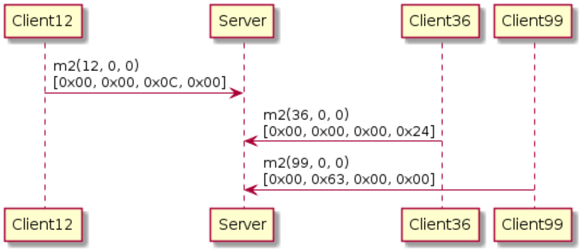

Figure 7: Impact of re-ordering parameters on three clients

3.3 Adding random parameters

This transformation adds a parameter to a message. Like the previous transformation, both the message definition and all the instances of `SendAction` referring to this message should also be updated. By default, the expression for this additional actual parameter is a call to a function returning a random value. This expression can however be customized to any valid ThingML expression, such as a constant or a call to another random function returning values in a sub-set of the original random.

3.4 Upscaling parameters

This transformation upscales a parameter within a message. By upscaling, we mean that the initial type of the parameter will be changed to a larger compatible type, if any. For example, a byte can be upscaled to an integer or a long, without losing information, and an integer can be upscaled to a long, and so on. For example, upscaling the parameter `b` of message `m2` to an integer, would produce the same result as parameter `b` (of type integer) of message `m1` in Figure 5 *i.e.*, it would be serialized using two bytes instead of one byte, shifting all the following bytes.

3.5 Duplicating messages

This transformation first clones a given message *m*, and the resulting new message *mBis* is renamed to avoid avoid any name confusion with the original one. The idea is to sometimes send the original message, and some other times send this new cloned message.

At this point, it is important to note that those individual transformations can be combined together, so that a given message can be transformed multiple times, possibly by different transformations. Even though the original and cloned message are very similar at the time when this transformation is applied, they can eventually become very different.

After the message has been cloned, two additional steps, illustrated in Figure 8, are performed:

- For all instances *sa* of `SendAction` referring to the original message *m*:
  - *sa* is cloned into a new instance of `SendAction` *sa'*, and *sa'* is updated so as to refer to the cloned message *mBis*.
  - a choice is made to determine when to send the original message and when to send the cloned message:
    - In *dynamic mode*, a new conditional action *ca* is created, *sa* being affected to the `then` clause of *ca* and *sa'* to the `else` clause. A new Boolean expression is created and

affected to *ca* to determine which branch to choose at runtime, based on a random threshold.

∗ In *static mode*, this choice is made by the diversifier so that a given instance will always either send the original message or the clone message. However, two instances duplicating the same message in the same way will not necessarily implement the same choice.

- For all transitions *t* reacting on the original message *m*, a transition *t'* is created by cloning *t* and updated so as it reacts on the cloned message *mBis*. Note that the cloning operation ensures that the source state, the target state, the guard and the actions are similar in *t* and *t'*, hence preserving the original execution semantics.

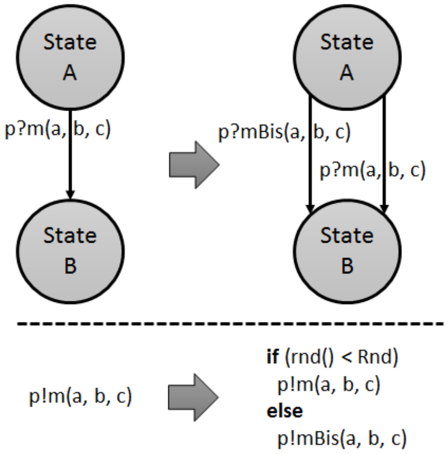

Figure 8: Duplicating Message: Before and After

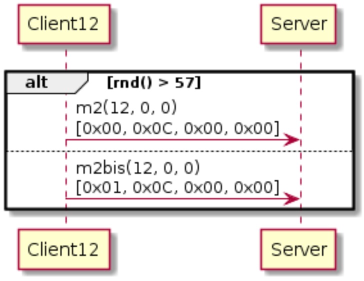

Figure 9: Impact of duplicating a message

Figure 9 shows the result of this transformation applied in dynamic mode for message m2. At runtime, whenever the client originally sent this message, it will now choose whether it should send this original message or the cloned message, based on the evaluation of a random function and a comparison to a threshold. This threshold is randomly chosen by the diversifier, so that different applications of the split message transformation will result in different distributions of these messages over time.

3.6 Splitting messages

This transformation is rather similar to the previous one in the way it is implemented, but is semantically quite different as shown in Figure 10:

- When applying the previous transformation, both the original and the new message are interchangeable. Whenever the original message was sent, the updated model can decide to send one or the other message, but not both. Whenever the original model was waiting for the original message, the updated model waits for one or the other message, but does not need to, and will not, wait for both messages.
- When applying this transformation, the original message is no longer used, and is replaced by two complementary, and non equivalent messages. Whenever the original was sent, the updated model must send both messages. Whenever the original model was waiting for the original message, the updated model waits for one and then the other message, in any order. An additional region is created in the state machine to handle this synchronization. When both messages are received, it will emit the original message on an internal port. The transition originally reacting on p?m(...) *i.e.*, a message m coming on an external port p is updated so as to expect this message on the internal port i.

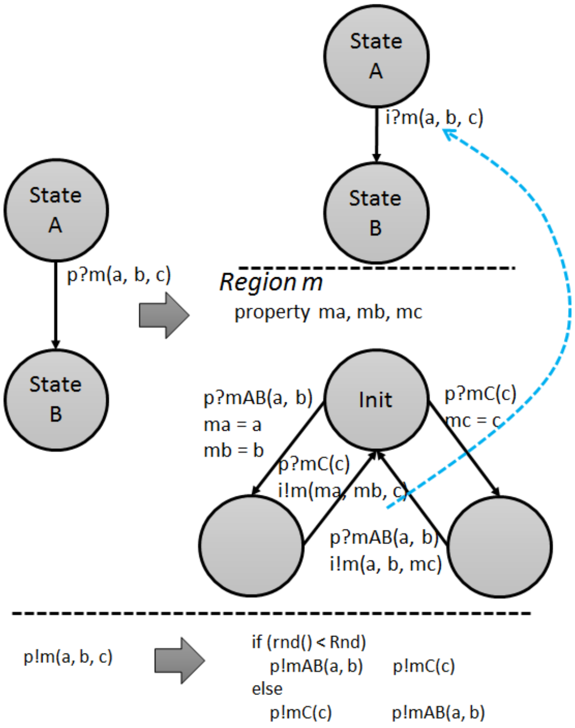

Figure 10: Splitting Message: Before and After

Figure 11 shows the result of this transformation applied in dynamic mode for message m2. At runtime, whenever the client originally sent this message, it will now choose in which order it will send both fragments resulting from the split, based on the evaluation of a random function and a comparison to a threshold. Again, this threshold is randomly chosen by the diversifier.

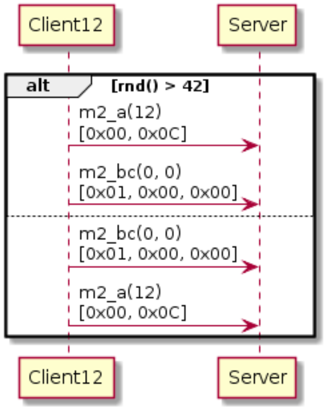

Figure 11: Impact of duplicating a message

4 VALIDATION

In this section, we evaluate our diversification approach, with respect to two dimensions:

(1) Assess the diversity of communications (Section 4.2), where we evaluate the quality of the diversity introduced automatically by our approach.
(2) Assess the overhead of diversified communications compared to non-diversified ones (Section 4.3) on a set of representative platforms.

4.1 Experimental Setup

In our experiments, we reused the basic protocol detailed in Section 2, composed of a client and a server exchanging 3 messages. This protocol was diversified to produce 100 statically-diversified protocols and 100 dynamically-diversified protocols.

The base protocol was run 100 times to simulate different clients, and the generated 200 diversified protocols were run once each. Each of those 300 protocols is executed on 3 platforms (JavaScript, Go and Arduino[5]) during 100 interactions. We collected all the raw bytes exchanged by the clients and server, as well as performance data (execution time, memory used, amount of bytes exchanges and size of the binaries) for those 90,000 interactions.

In addition to the binary data, we also simulated a "weak" parameter in one of the messages, and recorded its position and type in the exchanged messages. This parameter could for example be a parameter used by attackers to force the system into a vulnerable mode *e.g.*, by provoking a buffer overflow [13, 14]

The base protocol is simple, by design, for the purpose of this experiment. Indeed, it is important to understand that the more complex the base protocol is, the more opportunities we have to introduce diversity. In other words, a simple base protocol corresponds to the worst case for our approach.

4.2 Assessing the Diversity of Communications

4.2.1 Qualitative analysis of the raw data. Figure 12 visualizes a sub-set of the raw bytes transferred between the client and server over 5 interactions (15 messages), for 10 instances of the base protocol compared to 10 instances of the dynamically diversified protocols. The clearly visible vertical stripes in the non-diversified data, demonstrate the repetition of data over time, and similarity across instances. These bytes correspond to the *ID* of the client, which explains why they stay the same over time, but are different across instances. In the diversified data however, it is apparent that both the structure of the messages and the amount of data transmitted for each instance vary significantly. Looking closer at the data from the first diversified instance, we also observe that the *ID* (bright rectangles) is not always transmitted at the same place during each interaction, indicating that the protocol is changing over time.

It is important to note that all the diversified clients reached the `STOP` state, showing that they were able to successfully communicate back and forth with their associated diversified server.

4.2.2 Quantitative analysis of the diversity. To evaluate the generated diversity in a quantitative manner, Figures 13 and 14 show the number of bytes that are different in the exchange between client and server, across instances and interactions respectively.

Figure 13 compares each instance to the others, for each mode. Each point along the horizontal axis shows the average number of bytes that are different to the other instances, divided by the number of interactions. Both static and dynamic diversifications introduce a significant amount of diversity, both for individual instances (thin bright lines) and on average (thick lines).

Figure 14 compares each interaction to the others. Each point along the horizontal axis displays the average number of bytes that are different to the other interactions, divided by the number of instances. The results indicate that both static and dynamic diversity introduce diversity for all individual instances, although the latter significantly more than the former. Interestingly, the static diversification also introduces diversity across interactions (*i.e.* along time). This can be explained by the *upscaling* of message parameters, yielding a higher impact on the underlying bytes.

4.2.3 Practical implications of diversity. Figure 15 offers a different view over the raw data visualized in Figure 12: only the simulated "weak" parameter from a single message is displayed. This highlights the impact of diversification on the data that an attacker would be interested in. The results show that the diversified protocol drastically changes the layout of the data, as visualized by changes in position and type of the parameter, compared to the non-diversified protocol where the parameter stays in the same place and with the same type both across instances and interactions.

> Overall, our model-driven approach to diversify communications is able to automatically introduce significant diversity along the space dimension (for different clients) and along the time dimension (for multiple interactions of a given client). All the 60,000 diversified interactions were successful and allowed each client to communicate with their respective server-side protocol.

[5] A dialect of C++ for AVR microcontrollers.

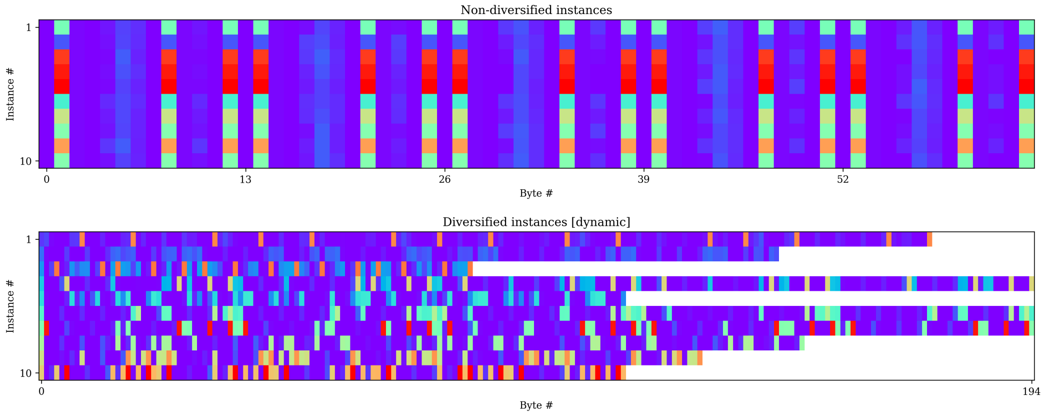

Figure 12: Raw data of non-diversified and dynamically diversified protocols for 10 instances and 5 interactions. The colors indicate the actual byte values. White sections correspond to no data (caused by differences in message lengths).

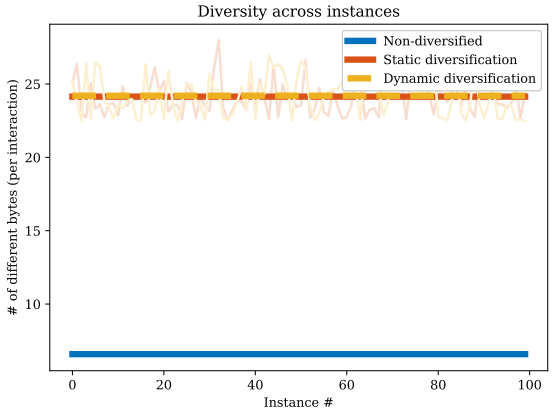

Figure 13: Analysis of the diversity across instances.

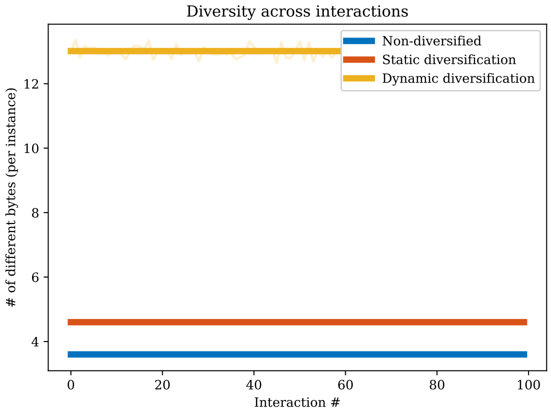

Figure 14: Analysis of the diversity across interactions.

4.3 Assessing the Overhead of Diversity

Table 1 summarizes the results of the benchmarks we conducted on 900 protocols: a) 100 base protocols, b) 100 statically-diversified protocols, and c) 100 dynamically-diversified protocols, were executed on Arduino, Go and JavaScript. More precisely, the table presents the overhead of b) and c), compared to the baseline performances obtained in a) for the three platforms.

Overall, the overhead of diversity is very well contained for JavaScript and Go. As those two languages are typically used on powerful machines (with several GB of RAM and storage, and several GHz CPUs), the overhead of diversity is almost imperceptible in practice. Indeed, beyond the communication logic, the business logic itself will use a significant amount of time and resources: it will check messages, persist some data, do some computing to come up with a response, and so on.

We highlighted in the table the results deserving a closer attention, which we further analyze in the remainder of this section.

4.3.1 Overhead on Arduino. Note that on Arduino, the 100 base protocols always use a fixed amount of resources. This is to be expected from a microcontroller running bare-metal (without OS) where any given instruction will always execute predictably.

Even though the RAM is extremely limited (2kB), the memory overhead is reasonably well contained.

The overhead on the binary size, especially in dynamic mode is rather high, compared to the 32KB of flash memory available on the Arduino. This is explained by the fact that the Arduino has no built-in hardware random generator, and a software random generator needs to be loaded in the dynamic version. This alone has however a limited impact (+0.5kb). Some overhead is related to the additional `if/else` statements that are inserted in dynamic mode to allow the Arduino to make choices at runtime on which messages to use (introduced by the duplicate and split transformations). We discuss in future work (Section 6) some strategies to reduce this overhead.

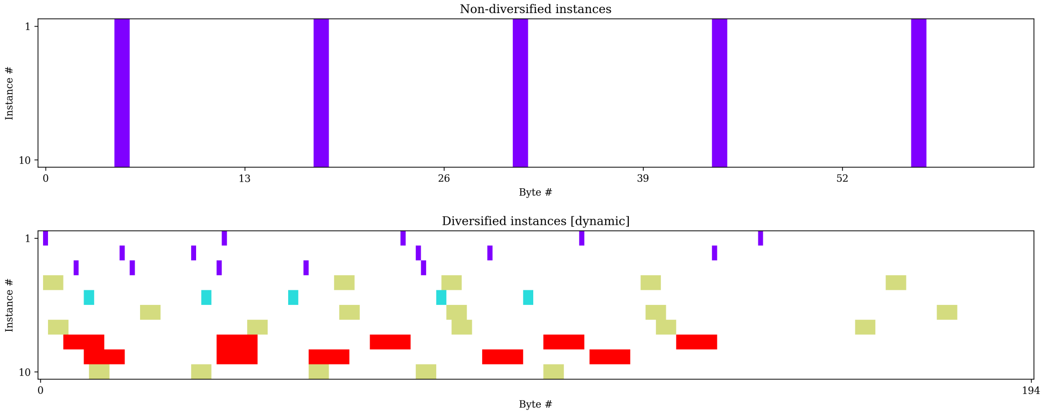

Figure 15: Simulated "weak" parameter: non-diversified versus diversified protocols. The panels show the same experiment as in Figure 12, while displaying only a single parameter. The colors corresponds to data type (byte, int, float, etc.).

Language	Mode	Time (per msg.)	Data (per msg.)	Memory	Binary size
JavaScript	Base	61.90 ± 9.40 µs	4.33 B	3.05 ± 0.04 MB	26.0 kB
	Static	+48.2 ± 12.0 µs	+2.97 ± 0.86 B	+148.0 ± 28.0 kB	+6.47 ± 0.02 kB
	Dynamic	+79.2 ± 19.6 µs	+3.10 ± 0.90 B	+295.0 ± 56.8 kB	+18.50 ± 0.02 kB
Go	Base	2.72 ± 2.42 µs	4.33 B	116.00 ± 0.27 kB	2.16 MB
	Static	+1.75 ± 1.65 µs	+2.97 ± 0.86 B	+19.10 ± 2.41 kB	+108.00 ± 3.84 kB
	Dynamic	+1.94 ± 1.76 µs	+3.10 ± 0.90 B	+20.30 ± 2.17 kB	+227.00 ± 5.81 kB
Arduino	Base	463 µs	4.33 B	697 kB	13.4 kB
	Static	+809.0 ± 74.1 µs	+2.97 ± 0.86 B	+76.70 ± 8.39 B	+8.09 ± 0.40 kB
	Dynamic	+1.04 ± 0.08 ms	+3.10 ± 0.90 B	+146.0 ± 13.7 B	+24.00 ± 1.01 kB

Table 1: Overhead

The overhead on the execution time is rather high on the Arduino compared to the to other platforms, though it evolves similarly than JavaScript and Go. The extra 200 µs in dynamic mode compared to the static mode can largely be explained by the software random generator and by the extra logic induced by the additional `if/else` statements. The overhead of 800 ms for the static diversity is important. Some initial investigations show that the additional regions created in the state machine by the split transformation have a significant impact on execution time. We will discuss in future work a possible refactoring for the split transformation, which is likely to reduce the execution overhead, and possibly helps reducing the binary size. Other platforms would also benefit from this refactoring.

This 1ms execution overhead can be an issue, in particular in safety-critical real-time control systems. Remember however, that our approach is targeted at "mass-produced" communicating software. For a device like the Arduino, this means typical Internet-of-Things applications, such a smart homes. If a smart-bulb needs to exchange 5-10 messages with the gateway before turning on or off the light, this would imply a 5-10ms longer delay in the actuation. For many similar applications where acceptable latency is in the order of 100ms or more, the overhead induced by our approach on execution time is negligible.

4.3.2 Constant data overhead across platforms. The overhead on the number of transmitted bytes (approximately +3 bytes per message) may seem suspiciously identical for the different platforms. This can however be simply explained. The 100 static protocols and the 100 dynamic protocols are the very same for the different platforms. As no choice is made at runtime in the static version, this trivially explains why the static modes are identical for all platforms. As for the dynamic diversity, it may still seem counter-intuitive that different platforms yield exactly the same amount of bytes on the network. Even though the choices made at runtime are most likely different for the different platforms, a closer look at Figures 9 and 11 reveals that no matter which choice is made, the amount of transmitted bytes will be the same, only the order and/or the value of the bytes will vary.

Overall, our model-driven approach to diversify communications only incurs a very contained overhead at runtime. This overhead is fully compatible with most use-cases where "mass-produced" software systems are used. On resource-constrained devices (8-bit microcontrollers), this overhead is significantly higher than on more powerful platforms, but still completely acceptable for a large number of IoT use cases.

5 RELATED WORK

The automatic introduction of diversity at different levels in the software stack is a promising way to increase the security and resilience of software systems [1, 4].

At runtime, Address Space Layout Randomization [15, 16] is an example of low-level diversity commonly found in modern operating systems. This diversification technique can mitigate certain runtime attacks by randomizing the places where programs and their data are loaded in RAM. Also at the OS level, Rauti *et al.* [17] propose a diversification framework for protecting operating systems, in particular focusing on diversifying the low-level system API provided by the OS. They showed, by experimenting on the Linux kernel, that this significantly impacts the graph call of applications interacting with those low-level system APIs. At this level, diversification techniques are generic and apply to any binary program. While such approaches can be beneficial, our model-driven approach to diversification weaves diversity into each and every application, fundamentally changing the way each instance behaves and communicates. Nevertheless, our approach benefits from, and is actually amplified by, approaches working at a lower level.

At the code level, numerous approaches propose to diversify code, for example code obfuscation [18, 19], to make it harder to decompile binaries and to reduce the ability to learn from decompiled binaries. Our approach does not intend to make each individual protocol harder to understand, but rather focuses on making each individual protocol differ, so as to make it harder to generalize from one example and learn from the overall system. Again, though our approach has a different focus, the code ultimately generated by our tool-chain can be obfuscated by these approaches. Diversification is a key principle for mutation testing [20, 21], where bugs are introduced by mutators in order to assess if existing tests can detect them. Though the underlying techniques are related, our approach produces semantically equivalent implementations. More related to our work, Cohen [22] proposed to protect operating system through evolving programs, by combining a set of mutations that transform instructions into different, yet functionally equivalent instructions. Similarly, Koo *et al.* [3] propose a binary re-writer implemented by extending the LLVM tool-chain by introducing fine-grained transformations. Conceptually, our work is similar, applying a set of model transformations to introduce diversity. However, the scope is different as 1) we specifically target communications, which to the best of our knowledge has not been investigated before, and 2) we introduce diversity at an abstract level, allowing us to target a wide range of languages and platforms, through a generative approach.

At the protocol level (as in TCP/IP), a few approaches propose to increase security through obfuscation or diversification [23]. Again, such approaches are related to our work but serve another purpose. More related to our work is the preliminary approach by Hosseinzadeh *et al.* [24], which propose to secure the Internet-of-Things (IoT) through obfuscation and diversification, leveraging the inherent diversity available in the IoT [25], for example different combination of the OSI network stack: HTTP, MQTT or CoAP for the application layer, IPv6 or 6LowPAN for the network layer.

Specific libraries also integrate diversity to increase their resilience. For example, software countermeasures, are usually added as a protection on top of cryptographic algorithms, to prevent (or make it less feasible) key recovery attacks [26]. FrankenSSL [27] is a recent approach recombining existing implementations of SSL-/TLS, the security mechanism behind HTTPS, to produce unique implementations of SSL and mitigate the risks of a successful attack targeting a given implementation. Those approaches relate to N-Version programming [28], where independent teams implement independent versions of the same design. Our approach also leverages the design of the application to introduce diversity, but aims at automating the diversification to the same extent than the generic approaches we previously commented.

6 CONCLUSION AND FUTURE WORK

This work introduces a new model-driven approach for the systematic diversification of communications between different entities. Diversification is introduced into abstract protocols through the combination of six atomic model transformations, and fully operational code is then automatically derived, covering a large number of platforms and network protocols. By analyzing 90,000 interactions, we showed that the diversity we automatically introduce produces significantly different protocols, contributing to reducing the learnability of the overall system. We also showed that the overhead induced by our diversification approach is very well contained and compatible with the requirements of most "mass-produced" software such as mobile or web applications communicating with a cloud server, or Internet-of-Things devices communicating with a gateway. On powerful devices (gateways and servers), this overhead will practically be imperceptible: less than 100 µs additional latency and less than 300 kB memory overhead. On 8-bit microcontrollers, the overhead is more significant (1 ms additional latency) but remains acceptable.

A remarkable feature of our approach is, being a model-driven approach, that existing diversification approaches operating at lower levels *e.g.*, in compilers or operating systems, can be applied to the code we generate and contribute to amplifying the effects of our approach. Beyond our approach, we see MDE as a convincing paradigm to manage and introduce rich software diversity by a) reasoning at a higher level of abstraction, b) understanding and relying on the semantics of programs, and c) automatically impacting the whole software stack through automated code generation.

In future work, we will improve our approach by:

(1) Reducing the overhead, in particular for resource-constrained microcontrollers, by inlining the split transformation directly in the original state machine, rather than creating a new region for each message we split. We believe this will significantly reduce the execution time overhead and reduce the size of the binaries.
(2) Allowing for more flexible diversification workflows. Right now, all transformations are systematically applied to all messages composing the protocol. We believe a rich diversity can still be achieved by only applying a different sub-set of transformations to each message. This will also contribute to reduce the overhead of our approach.

We will also generalize our MDE approach to diversify more aspects of software, for example to diversify the control flows and data flows within each computing node of a distributed system.

REFERENCES

[1] Benoit Baudry and Martin Monperrus. The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond. *ACM Comput. Surv.*, 48(1):16:1–16:26, September 2015.

[2] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated Software Diversity. In *IEEE Symposium on Security and Privacy*, volume 10, 05 2014.

[3] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis. Compiler-assisted Code Randomization. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 472–488, 2018.

[4] S. Hosseinzadeh, S. Hyrynsalmi, M. Conti, and V. LeppÃďnen. Security and Privacy in Cloud Computing via Obfuscation and Diversification: A Survey. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 529–535, Nov 2015.

[5] Dusko Pavlovic. Gaming security by obscurity. In *Proceedings of the 2011 new security paradigms workshop*, pages 125–140. ACM, 2011.

[6] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. DDoS in the IoT: Mirai and other botnets. *Computer*, 50(7):80–84, 2017.

[7] Sicong Shao, Cihan Tunc, Pratik Satam, and Salim Hariri. Real-Time IRC Threat Detection Framework. In *Foundations and Applications of Self* Systems (FAS* W), 2017 IEEE 2nd International Workshops on*, pages 318–323. IEEE, 2017.

[8] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in WPA2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1313–1328. ACM, 2017.

[9] Brice Morin, Nicolas Harrand, and Franck Fleurey. Model-based Software Engineering to tame the IoT Jungle. *IEEE Software*, 34(1):30–36, 2017.

[10] Hamed Okhravi, Thomas Hobson, David Bigelow, and William Streilein. Finding Focus in the Blur of Moving-Target Techniques. *Security Privacy, IEEE*, 12(2):16–26, Mar 2014.

[11] Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. ThingML: a language and code generation framework for heterogeneous targets. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 125–135. ACM, 2016.

[12] Franck Fleurey, Brice Morin, Arnor Solberg, and Olivier Barais. MDE to manage communications with and between resource-constrained systems. In *International Conference on Model Driven Engineering Languages and Systems*, pages 349–363. Springer, 2011.

[13] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[14] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard TM: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*, volume 12, pages 91–104, 2003.

[15] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[16] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[17] Sampsa Rauti, Johannes Holvitie, and Ville Leppänen. Towards a diversification framework for operating system protection. In *Proceedings of the 15th International Conference on Computer Systems and Technologies*, pages 286–293. ACM, 2014.

[18] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM, 2003.

[19] Igor V Popov, Saumya K Debray, and Gregory R Andrews. Binary Obfuscation Using Signals. In *USENIX Security Symposium*, pages 275–290, 2007.

[20] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2011.

[21] Fevzi Belli, Christof J Budnik, Axel Hollmann, Tugkan Tuglular, and W Eric Wong. Model-based mutation testingâĂŤapproach and case studies. *Science of Computer Programming*, 120:25–48, 2016.

[22] Frederick B Cohen. Operating system protection through program evolution. *Computers & Security*, 12(6):565–584, 1993.

[23] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. Skypemorph: Protocol obfuscation for tor bridges. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 97–108. ACM, 2012.

[24] Shohreh Hosseinzadeh, Sampsa Rauti, Sami Hyrynsalmi, and Ville Leppänen. Security in the internet of things through obfuscation and diversification. In *Computing, Communication and Security (ICCCS), 2015 International Conference on*, pages 1–5. IEEE, 2015.

[25] Brice Morin, Franck Fleurey, and Olivier Barais. Taming heterogeneity and distribution in SCPS. In *Proceedings of the First International Workshop on Software Engineering for Smart Cyber-Physical Systems*, pages 40–43. IEEE Press, 2015.

[26] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, and Martin Jepsen. Analysis of software countermeasures for whitebox encryption. *IACR Transactions on Symmetric Cryptology*, 2017(1):307–328, 2017.

[27] Bheesham Persaud, Borke Obada-Obieh, Nilofar Mansourzadeh, Ashley Moni, and Anil Somayaji. Frankenssl: Recombining cryptographic libraries for software diversity. In *Proceedings of the 11th Annual Symposium On Information Assurance. NYS Cyber Security Conference*, pages 19–25, 2016.

[28] Algirdas Avizienis. The N-version approach to fault-tolerant software. *IEEE Transactions on software engineering*, (12):1491–1501, 1985.