Deliverable reference:	Date:	Version:	Responsible partner: UIO
D1.1	09.11.2017	V1.0	

DiversIoT

Project co-funded by the Norwegian Research Council
IKTPluss

http://its-wiki.no/wiki/DiversIoT:Home

Title:

D1.1 Survey of Security Diversification Mechanism for IoT

Editor(s):	Approved by:
Christian Johansen	Project Technical Manager: Arnor Solberg
	Classification: Unrestricted

Abstract / Executive summary:

This survey was published as Technical Report 473 by the Institute of Informatics, University of Oslo, October 2017. It is attached in full as this deliverable.

The work was presented on 1-3 Nov in Turku, Finland at the 29th edition of Nordic Workshop on Programming Theory (NWPT'17) by Shukun Tokas.

The final version of this survey is planned to be submitted to Elsevier Journal of Logical and Algebraic Methods in Programming (a Norwegian level 2 journal).

Document URL:	ISBN:	The Research Council of Norway With funding from The Research Council of Norway

Content

Content 2
Version History 4
1 Introduction 5
Technical Report Attachment follows. 6

Table of Figures

NO TABLE OF FIGURES ENTRIES FOUND.

Version History

Version	Description	Date	Who
0.1	First complete version		Brice Morin

1 Introduction

In this comprehensive study of diversification techniques for IoT we have made several concrete observation identifying open research questions. Two major general points could be summarized here.

(A) The field of diversification techniques is highly active with many recent surveys and results published in venues of highest ranking, like IEEE and ACM journals and conferences (e.g., ACM Computing Surveys or IEEE Symposium on Security and Privacy), and dedicated workshops of strong impact like Workshop on Moving Target Techniques. However, the techniques are usually developed for standard IT systems, i.e., with powerful operating systems or running in clouds or personal computers.
We found almost no works specially targeting IoT systems, let alone surveys or comprehensive studies of implementations for IoT systems (i.e., where results like computation, memory, usability would be
studied). Therefore, we see this study as timely and the open questions as useful for the advancement of security in IoT.

(B) From the existing diversification techniques (also called moving target techniques) we gave an initial opinion on which can be more easily applied to IoT systems, and which not. However, for those that we think are applicable more research is needed to perform thorough studies of feasibility and usability, both because of the constrained nature of the IoT devices (so not all heavy computations are feasible) and also because of the software development and business/feature requirements on IoT systems (for which we expect only the most easy to use methods would gain wide adoption). Still, there are many security and privacy aspects that we see in standard IT systems which could be useful for diversification. These could be applied at various levels and targeting different security and privacy requirements.
We also identify new possible lines of diversification techniques which could be developed starting from modern programming languages, like the one we proposed based on modern concurrent programming languages, like the popular GoLang or the object-oriented Creol, or modelling languages like Statecharts or ThingML.

Technical Report Attachment follows.

UiO : Department of Informatics
University of Oslo

Survey of Security Diversification Mechanisms for Internet of Things (long version)[1]

Shukun Tokas , Olaf Owe , Christian Johansen

Research report 473, October 2017

ISBN 978-82-7368-438-7
ISSN 0806-3036

Abstract

Internet of Things (IoT) is the networking of physical objects having embedded various forms of electronics, software, and sensors, and equipped with connectivity to enable the exchange of information. The IoT wave is considered as the next stage in "the information revolution". IoT is gaining popularity for the great benefits it can offer in domestic and industrial settings as well as public infrastructures. However, securing IoT ecosystems is a complex and daunting task, which is largely disregarded by industry for the reasons: *i)* IoT devices are inherently resource constrained, and *ii)* the need of IoT devices to be cheap is the business driving force that asks for functionality instead of safety and security. The poor security protection makes IoT systems more susceptible to exploiting vulnerabilities.Moreover, IoT devices are meant to be deployed in large numbers (in billlions) and in ways that make it difficult to upgrade. The fact that such a large amount of devices are programmed in the same way allows an attacker to exploit one vulnerability in millions of devices by repeated application of the same attack, thus with much more gains at the same cost. In this paper, we propose to address the challenges pertaining to security and robustness in IoT settings by deliberate *inclusion of diversity* in design and development of IoT devices. Moreover, the diversification mechanisms have to preserve the intended semantics of the overall application. First, we examine diversification mechanisms in several relevant domains, and then we identify mechanisms that could be applicable in IoT settings, in terms of feasibility, security, and performance overhead. Also, we propose a layered approach to program diversity by using concurrent programming to diversify program‘s observable behaviour.

[1] *Address for correspondence:*
Department of Informatics, University of Oslo, P.O. Box 1080 Blindern, 0316 Oslo, Norway.
E-mail: `{shukunt,olaf,cristi}@ifi.uio.no`

Contents

1 Introduction **3**

2 IoT Attack Landscape **5**
2.1 Memory Corruption 5
2.1.1 Memory Manipulation 6
2.1.2 Memory Leak 6
2.2 Binary Modifications 7
2.3 Timing Based Attacks 7

3 Security and Diversity **7**
3.1 Automated Software Diversity 9
3.2 Moving Target Defence Techniques 10

4 Related Work **10**

5 Diversity and IoT **12**

6 Layered approach to program diversity **17**

7 Conclusion **19**

1 Introduction

Mark Weiser first phrased the vision of ubiquitous computing in 1991 as "The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it"[1]. *Internet of Things* [2] is meant to support this ubiquitous computing wave. The term *IoT* was coined by the British technology pioneer Kevin Ashton [3], who described it as a system of physical world objects connected to the Internet via ubiquitous sensors. IoT is advancing fast towards smart homes, smart healthcare, smart cities, and other innovative applications meant to help people's daily lives, by automating lot of routine activites. The adoption of this paradigm in a smart infrastructure requires connecting - sensors, actuators, other computing devices etc - to the Internet using IoT enabling technologies. These *connected things* then cooperate, to achieve a common objective for certain infrastructure (e.g. autonomous light control in public facilities, energy efficient home, assisted living for elderly people etc).

However, majority of these IoT devices are resource constrained, with low computational capabilities and limited memory. These constraints leads to using less resource intensive mechanisms for storage, computation, encryption etc. IoT is driven by industry and as such is focused on functionality instead of safety or security, also because security mechanisms adds latency to application processing. Altogether, these are often the reasons that, often leads to poor security in IoT settings. Moreover, IoT systems have become quite simple to program and build, e.g., see various kits/DIY for children or hobbysts based on e.g., Arduino, RasberyPi, or Intel Edison. Additionally Because of this there is an explosion of poorly secured IoT systems.

As mentioned earlier, IoT devices are often deployed in personal space, public places, industrial setups etc and from these contexts they access and transmit *sensitive information*. Sensitive information could be personal identification numbers, health records, movement patterns etc. Since these intentionally monitor and control human environments, exploiting such systems can have more dire consequences than usually with PCs, like serious privacy breaches due to sensitive data from IoT sensors reaching wrong hands, or lethal safety consequences due to hackers taking control of critical human environment components like fire alarm, stairway lighting, etc. As a consequence, these breaches then leads to privacy violations, unlawful tracking, mass surveillance etc. Opposed to risks for persons, IoT brings new kinds of risks for businesses that are dealing with information. In recent times IoT devices have been used to launch major cyberattacks, such as denial-of-service attacks, on corporations. As an example, the Mirai botnet scanned the Internet for poorly secured IoT devices (including security cameras, baby

monitors, etc) infecting, and taking control of more than 100,000 used to orchestrate a massive DDoS (distributed denial-of-service) attack [4] by generating masked TCP/UDP packets from infected nodes to saturate network resources. The magnitude of the attack reached 1.2Tbit/s and made websites – Twitter, Paypal, Amazon, CNN and many more – inaccessible for users in US east coast and Europe, because their domain name provider, Dyn, was forced offline by this DDoS attack. Several other large-scale DDoS attacks that had happened recently, targeted KrebsOnSecurity.com, cloud provider OVH, and Deutsche Telekom.

In those cases, one of main reason for success of these large scale attacks was *ubiquity of devices* and presence of similar vulnerabilities in large number of these devices. In general, computing systems are designed and developed identically, for the economic benefits of mass production. It also has considerable benefits of consistent behavior, simplified distribution and maintenance. However, all these advantages becomes potential weakness when an attacker succeeds in exploiting a vulnerability in one device, the same exploit can be replicated and distributed to compromise other identical devices. Due to lack of diversity, the impact of such attacks grow with number of identical devices.

In 1997, Forrest et al. presented an analogy between "diversity in biological system"and "diversity in computer system", and also suggested its benefits of diversity in computing systems from security perspective. Diversity can be defined as, the condition or quality of being diverse, different, or varied. In ecology, *diversity-stability hypothesis* states that higher diversity within biological communities tends to increase resilience. By resilience we mean the ability to be resistant against attacks and the ability to recover quickly and with limited damages in case of infringements. This hypothesis can be utilised for building resilient IoT ecosystem.

"Huge number of connected devices and presence of same vulnerabilities in millions of devices"was the observation that motivated us to pursue inclusion of diversity in development of IoT devices for security gains.
In this paper, we make the following contribution:
i) explore relationship between *Diversity* and *Security*, ii) briefly describe existing diversification mechanisms in computing systems, iii) identify diversification mechanisms that can be userful in IoT settings, and iv) propose to use concurrent programming (such as Creol [5]) for the development of IoT devices, to utilise it's inherent *non-determinism*.

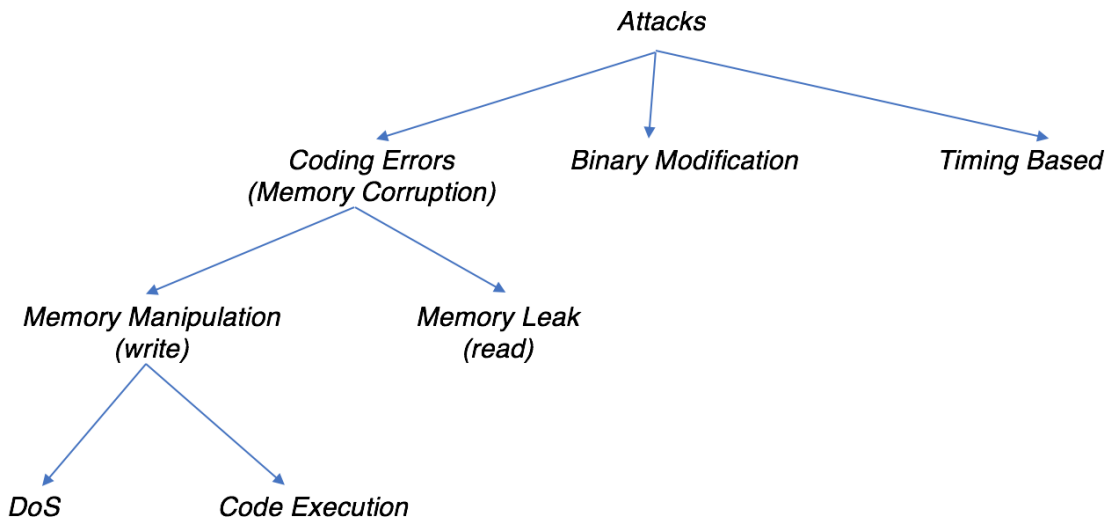

Figure 1: Attack Taxonomy

2 IoT Attack Landscape

To strengthen the security mechanisms and have an effective defence, it is essential to understand how an attacker operates. The attack taxonomy considered here, is a customized version of the Common Attack Pattern Enumeration and Classification (CAPEC) attack taxonomy. In this section, we briefly discuss some of prominent attacks in cyberspace that are also applicable in IoT settings. In later sections, to motivate the study of diversification mechanisms for security gains in IoT, we discuss certain diversification mechanisms that can be effective against these attacks.

2.1 Memory Corruption

Memory corruption, is a generic term, that is often used to describe unintended modification of the contents of virtual memory location in runtime, and due to this an unintended use-case of the program is executed. Programming errors are one of the most likely cause of memory corruption, with buffer overflow as a notable vulnerability. This also include other situations such as accessing uninitialised memory, using dangling pointers, incorrect pointer arithmetic etc.

However, buffer manipulation is one of the most commonly exploited vulnerability in IoT settings. According to statistics from National Vulner-

abilities Database [6], the number of buffer errors have increased from 723 to 1,916 (Jan 2012 - Sep 2017). For several reasons, such as performance or access to low-level constructs C/C++ remain a popular choice for development of embedded systems. Moreover, C/C++ does not provide sufficient protection against buffer overflow. Many memory manipulation functions (in C/C++) do not perform bound checks and can be used to read/write unintended bounds of buffer. Not performing the bound checks, or in other words not ensuring which memory locations are valid for the buffer, can lead to read and write operations to be performed on memory locations associated with other variables, or data structures. This can be exploited by the attacker to execute arbitrary code or read sensitive information. Also, corrupted memory contents leads to program crash or unintended program behavior.

Memory corruption attacks can further be considered to be composed of: memory manipulation (write) and memory leak (read) attacks. However, for both attacks the target program performs inadequate bounds-checking on the memory, or invalid object usage etc.

2.1.1 Memory Manipulation

While in memory manipulation attacks, attackers write beyond defined buffer-bounds to redirect program execution or cause program to crash. For example, an attacker can force malicious behaviour by *injecting source code* into the target program, and then the injected code is executed as legitimate code by the application. In case of modern exploitation, the attacker inject data that leads to code-reuse exploitation (e.g. return oriented program). In another example, malicious input can be used for constructing commands. A target application accept certain input from user and uses that input to construct a command, and then execute that command. Lack of proper input validation leads to inclusion of untrusted user supplied data (HTTP headers, cookies etc) in the command construction. Consequently, this results in execution of unintended commands with privileges of target vulnerable application.

2.1.2 Memory Leak

Memory leaks occurs when certain memory is not required by the application anymore, but yet not freed i.e. not returned to pool of free memory. As a consequence, these unwanted references (e.g. objects on heap) leads to reading (and consequently leaking) sensitive information.

2.2 Binary Modifications

One of the major security challenges with IoT applications is that of reverse engineering. To modify a particular binary, the attacker needs to understand what it does and how it does this. In the first step, binary is disassembled i.e. translate the binary into assembly instructions. And then, decompile the code obtained from previous step. This step basically involves trying to understand composing parts and finding patterns that can be translated into source code. And finally, this compact and high-level code is then analysed. As a result of successful analysis, the attackers can determine details of its design and implementation, and then use this knowledge to modify aplication's binary code. For example, the obtained information can be used to inject backdoor code that can reveal sensitive information such as cryptographic constants, ciphers, or user's sensitive information.

2.3 Timing Based Attacks

Timing based attacks are based on analysis of time taken by the system to respond to different queries and computations. Timing variations analysed over a significant period and combined with statistical analysis, can reveal user credentials including encryption keys. Considering context of cryptography, the execution time for a cryptographic operation, to an extent, depends on the key. Timing attacks were effective on popular implementation of RSA [7, 8, 9], and those implementations of RSA were also used on smartcards or embedded devices.

3 Security and Diversity

To create a successful exploit, an attacker needs to know (or sometimes able to predict easily) either the program layout or the position of a statements in code or both. Considering application's source code, the key idea of diversification is to randomize the place of statement in the code and/or the memory layout. And with each diversification the timing characteristic is always influenced.

As mentioned in earlier section, the intention of applying diversification mechanisms is to have certain security gains. For information systems, in general, the NIST standards [10] require that information security encompasses, at least, the properties of confidentiality, integrity, and availability. Other fundamental security attributes that are often needed are: authentication, access control, non-repudiation, secure bootstrapping, tamper-resistance for

devices, etc. Which essentially means, security aims at protecting information against unauthorised access and modification at all levels: storage, processing, and transit, maintaining a reliable access to information for authorised entities.

In industrial setups, complex and sophisticated security mechanisms are employed and often maintained by dedicated security team. But attackers still try to find vulnerabilities to exploit systems, with purpose to get access to intellectual assets or to destroy valuable assets. For example, in 2014 e-mail (yahoo! mail) service for 273 million users was hacked, and in the same year login credentials along with contact information of 233 million e-bay customers were compromised. Another, recent and IoT relevant example, also briefly mentioned earlier, is *mirai botnet attack*. Mirai botnet scanned the Internet for poorly secured IoT devices (including security cameras, baby monitors, etc), and then infected more than 100,000 of those devices. Infected devices were used to orchestrate a massive DDoS attack [4] by generating masked TCP/UDP packets from infected nodes to saturate network resources. The magnitude of the attack reached 1.2Tbit/s and made these websites: Twitter, Netflix, Paypal, Amazon, CNN and many more, inaccessible for users in US east coast and Europe, because their domain name provider, Dyn, was forced offline by the DDoS attack.

As a matter of fact, the vulnerability that mirai botnet exploited was: all the electronic boards (for DVRs and webcams, manufactured by XiongMai Technologies) had hardcoded, default username and password. As a result, success of mirai botnet attack can be attributed to, presence of same vulnerability in approximately a million of identical devices. And impact of such attacks can be quantified by the cost associated with *i)* unavailability of information/computing resource, and *ii)* theft of confidential information.

As has been noted, the computing systems are designed and developed identically, for the economic benefits of mass production and also to leverage benefits of consistent behavior and simplified maintenance etc. However, all these advantages becomes potential weakness when an attacker succeeds in tampering with one instance, and then automating and distributing this attack to run on other identical instances. The impact of attack grow with number of identical systems.

On the other hand, it is a well known fact that *diversity* is an important source of robustness in biological systems. For example, an ecosystem that includes multiple species that serve similar functions or roles, some of these species can survive in cases of natural disturbances such as disease or climate change. The analogy between "diversity in species"and "diversity in computing systems ", and the benefits of diversity in computing systems for security gains, was noticed by Forrest et al. [11]. Indeed, this is the main ob-

servation that motivates this work. Though, inclusion of diversity will come with overheads (runtime and/or compile time) causing some inefficiency, and will also increase development costs, but despite these challenges it also offers considerable benefit of making overall IoT ecosystem robust and secure, against automated attacks.

Having varied implementations with same functionality (i.e. diversification) will make attacker's task more complex, as diversification introduces randomness and consequently makes it complex for the attacker to predict the detailed behaviour of system. That being the case, exploiting one node does not save any efforts on compromising another node, thus diminishing attacker's success rate.

3.1 Automated Software Diversity

Software diversity is a research topic with several recent comprehensive surveys [12, 13, 14]. Diversity techniques can be simply summarized as introducing uncertainty in the targeted program. Detailed knowledge of the target software (i.e., the exact binary rather than the high level code) is essential for a wide range of attacks, like memory corruption attacks, including control injection [15, 16, 17, 18]. Diversity techniques strive to include in software implementations high entropy so the attacker has a hard time figuring out the exact internal functioning of the system. The range of techniques for diversification through program transformation is large, and include approaches that vary with respect to threat models, security, performance, and practicality [12].

The software design methodology *N-variant* is an example of automation that we want to achieve for software diversification. The need for n teams of developers developing n variants of the same software independently, from a common specification, should be replaced with automated techniques based on algorithms with mathematical guarantees (e.g., probabilistic or logical guarantees) that would produce the n variants from the same software specification, or implementation given by one team of developers (e.g., [19]).

Software diversification has been applied at all levels of software, reaching the microprocessors level [20], the compiler [21] or the network [22].

Automated techniques from programming languages like information flow static analysis [23] have been extended to the dynamic setting to protect against code injection. Dynamic taint analysis [24] automatically detects injection attacks without need for source code or special compilation for the monitored program, and hence works on commodity software. TaintCheck [24] is an example tool that can perform dynamic taint analysis by performing binary rewriting at run time. The technique was shown useful against Cross

site scripting attacks [25]. Such techniques are still very popular and have been e.g., adopted for mobile operating systems [26] to protect the privacy of mobile apps [27].

Automated software diversification can also be used to counter bugs in software at runtime, thus making the system more robust. Applications to embedded systems have been proposed [28].

3.2 Moving Target Defence Techniques

More recently the terminology *"moving target defence"* was adopted as an umbrella term for various diversification techniques [29, 30, 31], with several comprehensive surveys [32, 33] that discusses various diversity defenses with their respective strengths and weaknesses. A moving target defense, also referred to as moving target technique, refers to strategies and mechanisms that introduces randomness to increase complexity and costs for attackers in attempts to defend system. Okhravi et al. [33] categorized these defense mechanisms into five domains - data, software application, runtime environment, operating systems, hardware - according to their place within the execution stack. Some of well-known moving target defense techniques, to protect against different attacks include: N-version programming, SQL-rand (applying instruction set randomization to SQL), RandSys (applying system call instruction randomization) etc. However, listing all techŋniques is beyond the scope of this report and also considering resource constraints with IoT devices, we will describe some strategies, and then mention their respective performance overhead and applicability in IoT domain.

4 Related Work

The concept of diversity had been applied earlier to: machine descriptions, source code, software applications, networks etc, in attempts to achieve broad heterogeneity to strengthen defense against risks of *monoculture*[34]. Salamat et al. proposes an intrusion detection mechanism, *Multivariant Execution*[35], that executes several variants of the same program to detect an intrusion. All variants are generated by application of (a combination of) diversity mechanisms such as stack base randomization, system call number randomization, register randomization, function reordering etc. However, all variants synchronize at system call level i.e. when variants are executed they make same system calls. Then, this multivariant execution is monitored by an independent program, a *monitor*. Any inconsistency in system calls during execution indicates an attack. This mechanism diversifies platform

properties at run time by diversifying stack growth directions, system call numbers etc.
Oberheide et al. constructed and deployed an antivirus, CloudAV[36] , as an in-cloud network service which employs multiple diverse detection engines for malwares. CloudAV identifies suspicious files on end host using a lightweight host agent, which then sends the file to a network service to identify malicious content. The experiment involved a network service consisting of 10 antivirus engines and two behavioral detection engines, running in parallel. CloudAV was able to detect 98 percent of the malware (of 7220 malware samples), as opposed to detecting only 35 percent of malware when single antivirus was used on the same data set.
Williams et al. [37] applied diversity techniques to extend and modify software toolchain. Program run in a virtual machine, Strata, which can transform a program at run time by injecting code. Strata, examines program instructions and then translate them before they actually execute on host computer. It uses two strategies to apply diversity transformations: i) CSD or calling sequence diversity at compile time and ii) ISR or instruction set randomisation at run time to mitigate, return-to-libc and code injection respectively. Barrantes et al [38], proposes, a vitualization mechanism, Randomized Instruction Set Emulation (RISE) to run randomized code. It randomize instruction set at load time by XORing a random mask with every byte of program code (including libraries). At run time, the randomized code is XORed with same mask for code execution. Any code injection, will result in invalid execution. However, this defence is based on assumption that attacker can not access process memory. Both techniques [37, 38] relies on reliability and integrity of emulator.
Christodorescu et al. proposed an *end-to-end diversification* technique[39] in which diversification mechanisms are applied to randomize instruction sets, script API, reference names of stored data and other key program elements. These mechanisms were applied to the Javascript component and SQL DB component of a web application. However, authors also indicates that, a diversified program instance is a result of repeated application of a set of transformations. This technique has potential to protect application against both high-level as well as low-level code injection attacks.
Often, the sensors are deployed in unprotected environments and are vulnerable to physical attacks. Alarifi and Du[40], proposed program obfuscation (code and data) to alleviate reverse engineering attacks on sensor nodes. First, the data structure that stores secret key is obfuscated using hash functions followed by code obfuscation, and then control flow of code is randomized. This scheme was implemented on Mica2 sensors and different obfuscation methods are applied to each sensor. Though this diversification

can not protect a sensor for too long, but it assures to raise the bar for a successful large-scale attack. With moderate runtime overhead, this technique eliminates the possibility of compromising (with same exploit) more than one node, by ensuring attacker target each node individually.
Caballero et al. [41] studied effectiveness of diverse software implementations on routers, to evaluate overall robustness of a network against vulnerability exploits. They capture the effect of router failure in a topology using graph colouring algorithm and demonstrated that small degree of diversity can provide good robustness against simultaneous router failures.
Donnell et al. [22] demonstrated an integration of diversity and distributed coloring algorithm to prevent successful launch of large scale attacks. Software packages can be diversified using diversification mechanisms. Network topology is designed such that, it minimizes the number of neighbours running same software packages. Different colors are used to represent different software packages. Several coloring algorithms were used to find an optimal coloring solution, experimental results shows randomized hybrid and best choice hybrid algorithm produces an effective coloring. Then the nodes are clustered based on identified colors to minimize number of connected nodes with same color. This approach is an effective defense against a single attack, as it prevent propagation of attack to non-trivial number of nodes, but it can not prevent attacks against a single node.

5 Diversity and IoT

In this section, we present a brief summary of some of the diversification strategies that are relevant in IoT settings. The objective of applying diversity mechanisms in the development and operations of IoT devices is to achieve heterogeneity in such a way, that it makes certain aspects (design, implementation, and execution) less predictable to attackers. Each of these diversity techniques add some randomization to introduce *uncertainty* in each variant of the device or program. However, these alterations preserve semantics of the program i.e. diversity should not result in producing different program outputs.

For instance, the things or any other computing devices interact with other devices or Internet through an *Interface*. Interface is an intersection between the thing and environment, which describes what can be sent/received by the thing to/from environment. The structure and definition of interface is fixed according to communication standards. However, rest of the implementations and configurations inside thing be done using diverse mechanisms, while preserving the structure and semantics of the interface.

As has been noted from related work section, diversity techniques can be applied to any part of: hardware, operating system, software, communication protocols, security mechanisms and everything in between, to attain desired diversity in memory usage, performances, timing, and program flow. We identified several diversification mechanisms that appear to be, feasible at reasonable cost and incur moderate overhead, applicable in IoT settings.

1. **N-version approach**
   A system specification gives detailed description of how a system behaves without describing how that behaviour is designed and developed. This fact can be useful for developing a fault tolerant system. N-version approach essentially means that, following the same specification separate teams work independently to design and develop N-equivalent versions [42]. As vulnerabilities may result at design and implementation phase, the N-version approach to generate variants of same system specification by independent teams improves fault tolerance [42]. This offers diversity at design level as well as implementation level. Diverse sources of faults, leads to considerably transient effects. A larger direction might be combining this technique with an N-version programming technique.

   Applicability in IoT: Considering the resource constrained nature of IoT device, this approach appears to be applicable because the specification will be in accordance. Conversely, considering the huge number of versions required and each version incurs an overhead in terms of resources/budget for N-teams working on N-variants (where N is possibly tens of thousands), this doesn't appear to be reasonable if used as a standalone diversification strategy. But this technique can be combined with other diversification techniques to achieve an effective diversification.

2. **Address Space Layout Randomisation(ASLR)**
   ASLR is one of popular dynamic diversification techniques, used in majorly used operating systems such as Android, DragonFly BSD, iOS, Linux, Windows, OS X etc. It provides diversity through run-time randomization, by randomizing the base address for the code and data. It performs stack randomization by changing base address of the stack, which provides fine-grained randomization as functions and variables will be randomly placed. It also removes heap from the data section and places it in program memory.

   Applicability in IoT: By randomising the base addresses for code, stack,

and heap, it provides significant security benefits but only induces insignificant impact on the performance [43]. In a survey by [44], 70% of manufactures preferred Linux as OS of choice for IoT gateways. In [43] detter and mutschlechner, measures performance and entropy for widely used implementations of Linux. ASLR can be considered an essential feature to implement for securing key data areas of processes running on IoT devices. ASLR raise the bar for attackers by making it complicated to exploit buffer overflow and code-reuse vulnerabilities, to launch memory corruption attacks such as return oriented program, code injection, memory leak. Considering SPEC CPU2006 benchmark suite, which is widely used to evaluate the processor performance, the average impact of ASLR was 9% and 2% (on 32-bit and 64-bit implementations respectively) [45]. Considering reasonable overhead, and protection against major software exploits, makes it useful in IoT settings as well.

3. **System Call Randomization**
   System calls enables a program to enter kernel space to perform operations on input/output devices. Randomizing system call numbers diversifies system call interface between processes and processor [46]. Each system call is assigned a unique system call number, which is stored in system call table. System call dispatcher uses that number to determine which system call to invoke. An illustration of this, is depicted in RandSys [47], which uses a combination of ISR and ASLR technique. When a process is executed for the first time, it scans all system call invocations and their location in memory. Then using a secret key stored in kernel space, a randomization algorithm is applied on {original system call number, location of system call} to generate a "randomized system call number". This requires making changes to system call dispatcher, to decrypt system call numbers at runtime. Considering this randomization, if a program is compromised by injecting some code then system call dispatcher cannot de-randomize the system call due to wrong location address and the process will crash.

   Applicability in IoT: Overhead includes encrypting (or randomizing) and then decrypting the system calls, by system call dispatcher. Average impact of this implementation on program execution could be upto 20% [33]. Assuming that the kernel is safe, this technique (RandSys) can protect against memory manipulation attacks, such as code injection and control injection. This technique also appears to be useful in IoT settings

4. **Program Obfuscation**
   Code transformation techniques, transforms a source program P into a (functionally) equivalent program P' [48]. Objective is to make low-level semantics of programs, harder and more complex for attacker to comprehend, without affecting program's observable behaviour. However, to have effective security and diversity the obfuscated code should be difficult enough for analysis and to reverse engineer. Collberg et al [48], identified 4 main classes of transformation, for code obfuscation and data obfuscation: lexical transformation, control flow transformation, data flow transformation, and preventive transformation. These transformation involves renaming variable, altering control flow of program by using opaque predicates or graph flattening, or changing the data encoding etc. Also, some transformations applied on data uses hashing mechanisms and can be considered as one-way transformation [49]. Application of varied transformations, diversifies the code in terms of code space and execution timings, and then the obfuscated code is distributed to clients.

   Applicability in IoT: Benefit of this technique is that it can be automated to generate large number of code variants, in a platform independent manner (considering transformation at source code level). On the other hand, it does incur an added cost due to memory usage and execution cycles required to execute obfuscated code. Considering SPEC CPU benchmark suite, the average impact of program ofuscation techniques on performance is approximately 11% [50]. This technique is an effective defence against attacks based on reverse engineering and code tampering. Altogether, makes it very useful diversification technique in IoT settings.

5. **Adding No Operation Performed (NOP) Instructions**
   Non-functional code, such as NOP, can be inserted to generate delay in execution or to indicate some space reservation in program memory. A NOP instruction doesn't affect any register, other than program counter (PC), and consumes only one clock cycle to perform PC = PC + 1. NOP instruction can be inserted between instructions without changing program semantics. This NOP insertion generates diverse binaries and makes the program execution more unpredictable to the attackers as variants will have different execution statistics.

   Applicability in IoT: Overhead induced is proportional to number of

NOP instructions included in the code. It doesn't degrade systems performance significantly and can be combined with other diversification mechanisms to have an effective diversification strategy. It can also be used to detect control flow change due to instruction misalignment.

6. **Concurrent Programming**

   So far we have mentioned, several existing diversification techniques. However, its worth noting that without originally being intended for diversification, *concurrent programming* have randomization implications. We suggest to explore concurrency as a source of non-determinism for diversifying source code in IoT devices. In concurrent settings, several processes may execute concurrently. High level modelling language, Creol [5], can be used for concurrent programming. Processes in concurrent settings may interleave with one another, generating a huge number of interleavings, making the program execution *non-deterministic*. This large number of possible interleavings can be used to diversify program execution.

   Applicability in IoT: Concurrent programming provides timing diversity, i.e. different interleavings results in different execution sequences accessing different memory regions. This way it makes it complicated for the attacker to know or predict, which statement will execute next to launch an attack. However, the high-level Creol code can be translated to low-level code in two ways. One way is to have a schedular at the manufacturer side. Manufacturer can generate low level code for millions of variants and distribute them to IoT devices. This will change scheduling once and for all, yet each one diversified due to different interleaving. Another way is, if the device is strong enough, we have schedular built into the device, which will have different interleavings with every execution.

In addition, to develop an understanding of how to best use these diversification mechanisms, it is essential to have *metrics* to quantify the impact of these diversification mechanisms on *security*. For example, entropy metrics to analyze effectiveness of randomness (or uncertainty) introduced in the variants, generated by various diversification mechanisms.

Depending on the use case, these diversification mechanisms can be applied individually or as a composition (e.g. combining program obfuscation and system call randomization, or combining concurrent programming and program obfuscation etc). However, it is not very clear at this point how such

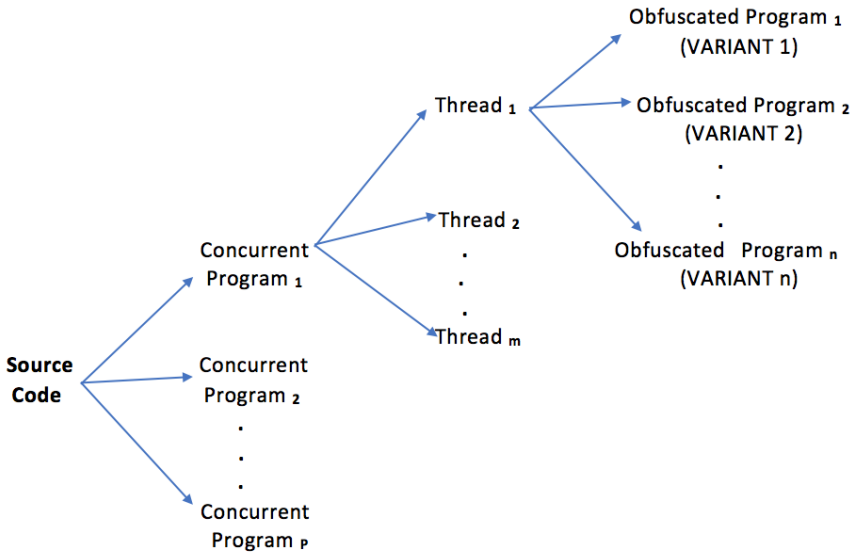

Figure 2: Managing Program Diversity.

selection should be made. And this can be considered as one of direction of future work.

6 Layered approach to program diversity

The idea is to combine diversification techniques and concurrent programming, to generate equivalent program variants that differ in memory usage, program flow, and execution timings. And the combination of these two for program diversity can be expressed as composed of two layers: *concurrent programming layer* and *program variant layer*. For the description of this approach, consider a source code P which is written as a sequential program, and P can also be designed as composed of m independent threads, i.e. $P = \{T_1, T_2 \ldots T_m\}$.

Concurrent Programming

We propose to structure the source code into multiple threads and interleave executions of these threads, either on single processor or multiprocessors. These threads communicate with each other via message passing. However, concurrent programs can be designed independent of single or multi processor execution environment, as the program exhibits non-determinism by the interleaved execution (in both cases). This technique is useful to pre-

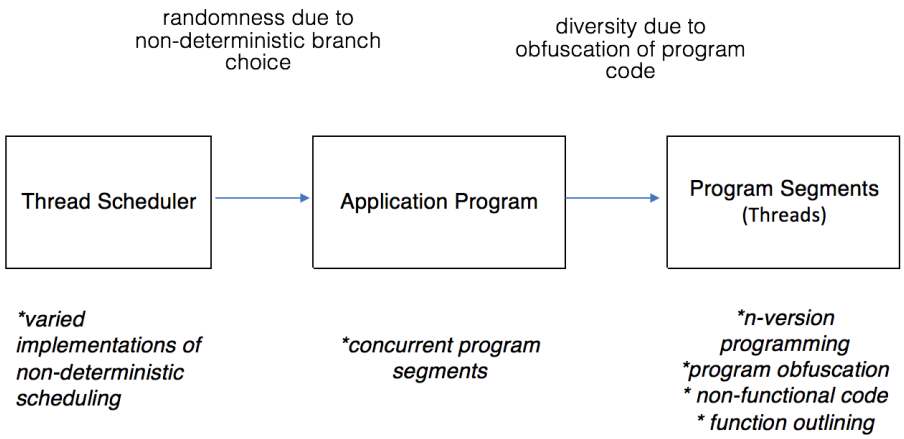

Figure 3: Layered Approach To Program Diversity.

vent against attacks based on knowledge of precise timing of events, because it becomes challenging for the attackers to observe when a program segment might be executing. However, it would be challenging for the program designers to determine which program segments to include in which threads and when these threads interact with each other, without affecting program correctness.

Program Variant

Applying diversification techniques on threads, as mentioned in previous section, functionally equivalent yet diverse variants can be generated. If application of certain diversification techniques can generate *n-variants* for one thread, then a source code P (with m threads) can have $m \times n$ variants. Also, programmer defined points of suspension adds to non-determinism, and can be explicitly added to the variants.

Inclusion of concurrent programming layer adds to the difficulty for attacker (especially for timing based attacks) because executing a sequential program reveals more information about program behaviour than executing an equivalent program as concurrent threads non-deterministically. This layered approach aids in managing diversity. For example, consider a source code P can designed (using concurrent programming) in p different ways, each program have m threads, and each thread can be implemented (applying diversification techniques) in n different ways. This way we have $p \times m \times n$ program variants.

7 Conclusion

In this comprehensive study of diversification techniques for IoT we have made several concrete observation identifying open research questions. Two major general points could be summarized here.

(A) The field of diversification techniques is highly active with many recent surveys and results published in venues of highest ranking, like IEEE and ACM journals and conferences (e.g., ACM Computing Surveys or IEEE Symposium on Security and Privacy), and dedicated workshops of strong impact like Workshop on Moving Target Techniques. However, the techniques are usually developed for standard IT systems, i.e., with powerful operating systems or running in clouds or personal computers. We found almost no works specially targeting IoT systems, let alone surveys or comprehensive studies of implementations for IoT systems (i.e., where results like computation, memory, usability would be studied). Therefore, we see this study as timely and the open questions as useful for the advancement of security in IoT.

(B) From the existing diversification techniques (also called moving target techniques) we gave an initial opinion on which can be more easily applied to IoT systems, and which not. However, for those that we think are applicable more research is needed to perform thorough studies of feasibility and usability, both because of the constrained nature of the IoT devices (so not all heavy computations are feasible) and also because of the software development and business/feature requirements on IoT systems (for which we expect only the most easy to use methods would gain wide adoption). Still, there are many security and privacy aspects that we see in standard IT systems which could be useful for diversification. These could be applied at various levels and targeting different security and privacy requirements.

We also identify new possible lines of diversification techniques which could be developed starting from modern programming languages, like the one we proposed based on modern concurrent programming languages, like the popular GoLang [51] or the object-oriented Creol [5], or modelling languages like Statecharts [52] or ThingML [53].

References

[1] M. Weiser, "The computer for the 21st century," *Scientific american*, vol. 265, no. 3, pp. 94–104, 1991.

[2] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[3] K. Ashton, "That internet of things thing," *RFiD Journal*, vol. 22, no. 7, 2011.

[4] T. Pultarova, "Webcam hack shows vulnerability of connected devices," *Engineering & Technology*, vol. 11, no. 11, pp. 10–10, 2016.

[5] E. B. Johnsen, O. Owe, and I. C. Yu, "Creol: A type-safe object-oriented model for distributed concurrent systems," *Theoretical Computer Science*, vol. 365, no. 1-2, pp. 23–66, 2006.

[6] "National vulnerability database." `https://nvd.nist.gov/`. Accessed: 2017-10-1.

[7] W. Schindler, "A timing attack against rsa with the chinese remainder theorem," in *CHES*, vol. 1965, pp. 109–124, Springer, 2000.

[8] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems, "A practical implementation of the timing attack," in *International Conference on Smart Card Research and Advanced Applications*, pp. 167–182, Springer, 1998.

[9] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Annual International Cryptology Conference*, pp. 104–113, Springer, 1996.

[10] R. Kissel, "Glossary of key information security terms," *NIST Interagency Reports NIST IR*, vol. 7298, no. 3, 2013.

[11] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*, pp. 67–72, IEEE, 1997.

[12] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *2014 IEEE Symposium on Security and Privacy*, pp. 276–291, IEEE, May 2014.

[13] S. Hosseinzadeh, S. Rauti, S. Laurén, J.-M. Mäkelä, J. Holvitie, S. Hyrynsalmi, and V. Leppänen, "A survey on aims and environments of diversification and obfuscation in software security," in *Proceedings of the 17th International Conference on Computer Systems and Technologies 2016*, CompSysTech '16, (New York, NY, USA), pp. 113–120, ACM, 2016.

[14] B. Baudry and M. Monperrus, "The multiple facets of software diversity: Recent developments in year 2000 and beyond," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, p. 16, 2015.

[15] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, (New York, NY, USA), pp. 30–40, ACM, 2011.

[16] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to risc," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, (New York, NY, USA), pp. 27–38, ACM, 2008.

[17] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *2012 IEEE Symposium on Security and Privacy*, pp. 601–615, IEEE, May 2012.

[18] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, pp. 2:1–2:34, Mar. 2012.

[19] 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 *USENIX Security Symposium*, pp. 105–120, 2006.

[20] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, "Core fusion: Accommodating software diversity in chip multiprocessors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, (New York, NY, USA), pp. 186–197, ACM, 2007.

[21] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz, *Compiler-Generated Software Diversity*, pp. 77–98. New York, NY: Springer New York, 2011.

[22] 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*, pp. 121–131, ACM, 2004.

[23] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, pp. 5–19, Jan 2003.

[24] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*, The Internet Society, 2005.

[25] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007*, The Internet Society, 2007.

[26] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, pp. 5:1–5:29, June 2014.

[27] M. L. Polla, F. Martinelli, and D. Sgandurra, "A survey on security for mobile devices," *IEEE Communications Surveys Tutorials*, vol. 15, no. 1, pp. 446–471, 2013.

[28] A. Höller, T. Rauter, J. Iber, and C. Kreiner, *Towards Dynamic Software Diversity for Resilient Redundant Embedded Systems*, pp. 16–30. Cham: Springer International Publishing, 2015.

[29] S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, eds., *Moving Target Defense - Creating Asymmetric Uncertainty for Cyber Threats*, vol. 54 of *Advances in Information Security*. Springer, 2011.

[30] S. Jajodia, A. K. Ghosh, V. S. Subrahmanian, V. Swarup, C. Wang, and X. S. Wang, eds., *Moving Target Defense II - Application of Game Theory and Adversarial Modeling*, vol. 100 of *Advances in Information Security*. Springer, 2013.

[31] S. Jajodia and K. Sun, eds., *Proceedings of the First ACM Workshop on Moving Target Defense, MTD '14, Scottsdale, Arizona, USA, November 7, 2014*, ACM, 2014.

[32] J. Xu, P. Guo, M. Zhao, R. F. Erbacher, M. Zhu, and P. Liu, "Comparing different moving target defense techniques," in *Proceedings of the First*

ACM Workshop on Moving Target Defense, MTD '14, (New York, NY, USA), pp. 97–107, ACM, 2014.

[33] H. Okhravi, T. Hobson, D. Bigelow, and W. Streilein, "Finding focus in the blur of moving-target techniques," *IEEE Security Privacy*, vol. 12, pp. 16–26, Mar 2014.

[34] B. Schneier, "The dangers of a software monoculture," *Information Security Magazine*, 2010.

[35] B. Salamat, T. Jackson, G. Wagner, C. Wimmer, and M. Franz, "Runtime defense against code injection attacks using replicated execution," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 4, pp. 588–601, 2011.

[36] J. Oberheide, E. Cooke, and F. Jahanian, "Cloudav: N-version antivirus in the network cloud.," in *USENIX Security Symposium*, pp. 91–106, 2008.

[37] D. Williams, W. Hu, J. W. Davidson, J. D. Hiser, J. C. Knight, and A. Nguyen-Tuong, "Security through diversity: Leveraging virtual machine technology," *IEEE Security & Privacy*, vol. 7, no. 1, 2009.

[38] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović, "Randomized instruction set emulation," *ACM Transactions on Information and System Security (TISSEC)*, vol. 8, no. 1, pp. 3–40, 2005.

[39] M. Christodorescu, M. Fredrikson, S. Jha, and J. Giffin, "End-to-end software diversification of internet services," *Moving Target Defense*, pp. 117–130, 2011.

[40] A. Alarifi and W. Du, "Diversify sensor nodes to improve resilience against node compromise," in *Proceedings of the fourth ACM workshop on Security of ad hoc and sensor networks*, pp. 101–112, ACM, 2006.

[41] J. Caballero, T. Kampouris, D. Song, and J. Wang, "Would diversity really increase the robustness of the routing infrastructure against software defects?," *Department of Electrical and Computing Engineering*, p. 40, 2008.

[42] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Transactions on software engineering*, no. 12, pp. 1491–1501, 1985.

[43] J. Detter and R. Mutschlechner, "Performance and entropy of various aslr implementations," 2015.

[44] "Capec-123: Buffer manipulation." `https://capec.mitre.org/data/definitions/123.html`. Accessed: 2017-09-30.

[45] M. Payer, "Too much pie is bad for performance," 2012.

[46] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *Security and Privacy (SP), 2014 IEEE Symposium on*, pp. 276–291, IEEE, 2014.

[47] X. Jiang, H. J. Wangz, D. Xu, and Y.-M. Wang, "Randsys: Thwarting code injection attacks with system service interface randomization," in *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, pp. 209–218, IEEE, 2007.

[48] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," tech. rep., Department of Computer Science, The University of Auckland, New Zealand, 1997.

[49] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in *Security and Privacy, 2009 30th IEEE Symposium on*, pp. 94–109, IEEE, 2009.

[50] C. Collberg, S. Martin, J. Myers, and J. Nagra, "Distributed application tamper detection via continuous software updates," in *Proceedings of the 28th Annual Computer Security Applications Conference*, pp. 319–328, ACM, 2012.

[51] A. A. A. Donovan and B. W. Kernighan, *The Go Programming Language*. Addison-Wesley, 2015.

[52] D. Harel and A. Naamad, "The STATEMATE semantics of statecharts," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 4, pp. 293–333, 1996.

[53] N. Harrand, F. Fleurey, B. Morin, and K. E. 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*, pp. 125–135, ACM, 2016.