The Journal of Systems & Software 200 (2023) 111633

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A model-based mode-switching framework based on security vulnerability scores ☆

Michael Riegler [a,c,*], Johannes Sametinger [a,c], Michael Vierhauser [a], Manuel Wimmer [b,c]

[a] LIT Secure and Correct Systems Lab, Johannes Kepler University Linz, Austria
[b] Christian Doppler Laboratory for Model-Integrated Smart Production, Johannes Kepler University Linz, Austria
[c] Institute of Business Informatics - Software Engineering, Johannes Kepler University Linz, Austria

ARTICLE INFO

Article history:
Received 6 December 2021
Received in revised form 3 October 2022
Accepted 4 February 2023
Available online 9 February 2023

Dataset link: Mode-Switching Framework

Keywords:
Mode switching
Security
Resilience
Domain-specific languages
Vulnerabilities

ABSTRACT

Software vulnerabilities can affect critical systems within an organization impacting processes, workflows, privacy, and safety. When a software vulnerability becomes known, affected systems are at risk until appropriate updates become available and eventually deployed. This period can last from a few days to several months, during which attackers can develop exploits and take advantage of the vulnerability. It is tedious and time-consuming to keep track of vulnerabilities manually and perform necessary actions to shut down, update, or modify systems. Vulnerabilities affect system components, such as a web server, but sometimes only target specific versions or component combinations.

In this paper, we propose a novel approach for automated mode switching of software systems to support system administrators in dealing with vulnerabilities and reducing the risk of exposure. We rely on model-driven techniques and use a multi-modal architecture to react to discovered vulnerabilities and provide automated contingency support. We have developed a dedicated domain-specific language to describe potential mitigation as mode switches. We have evaluated our approach with a web server case study, analyzing historical vulnerability data. Based on the vulnerabilities scores sum, we demonstrated that switching to less vulnerable modes reduced the attack surface in 98.9% of the analyzed time.

Editor's note: Open Science material was validated by the Journal of Systems and Software Open Science Board.

1. Introduction

Software weaknesses are "flaws, faults, bugs, vulnerabilities, and other errors in a software systems implementation, code, design, or architecture that, if left unaddressed, could result in systems and networks being vulnerable to attack" (MITRE, 2022c). Examples of such weaknesses include buffer overflows, improper certificate validation, or using components with known vulnerabilities. An information security vulnerability in turn is "a mistake in software that can be directly used by a hacker to gain access to a system or network" (MITRE, 2022b).

Such software vulnerabilities are often the entry point for attackers, providing a bigger attack surface for affected systems. As a result, systems are potentially exposed to attacks from the time when vulnerabilities become known to the time when they are finally patched. This window of exposure is critical, because it renders systems without protection, hence, this time should be as short as possible. To mitigate this problem, we can reduce the risk of being vulnerable to certain attacks or reduce the time in which the system exhibits the vulnerability (Schneier, 2000). However, keeping track of all potential vulnerabilities of a company's infrastructure is a tedious and time-consuming task. Vulnerabilities often affect specific components of a software system and often only specific versions (or a range of versions).

For example, Google's Project Zero reported a vulnerability in a stable release of the Apache Web Server (version 2.4.43) with technical details and a *Proof-of-Concept* (PoC) to the Apache security team on April 24, 2020 (Wilhelm, 2020). The vulnerability could cause memory corruption and lead to a crash and *Denial of Service* (DoS). Apache analyzed the issue, and 105 days later, they provided an update for release 2.4.44 on August 7 (Apache, 2020). On the same day, the vulnerability was published on the *US National Vulnerability Database* (NVD) as CVE-2020-9490 with a severity of 7.5 (out of 10) (NIST, 2020). In this context,

☆ Editor: Matthias Galster.

* Corresponding author at: LIT Secure and Correct Systems Lab, Johannes Kepler University Linz, Austria.
E-mail addresses: michael.riegler@jku.at (M. Riegler), johannes.sametinger@jku.at (J. Sametinger), michael.vierhauser@jku.at (M. Vierhauser), manuel.wimmer@jku.at (M. Wimmer).

https://doi.org/10.1016/j.jss.2023.111633

Common Vulnerabilities and Exposures (CVEs) provide an identification number, a description, and references for publicly known vulnerabilities (MITRE, 2022b). However, the window of exposure is not yet closed with the vendor patch. For example, it took the Debian package maintainers another 18 days to backport and fix this issue on August 25 in their repositories for the stable distribution Buster (Debian, 2020). After that, system owners can install the patch. In total, the software was vulnerable for at least 123 days. On December 7, a copy of the PoC was published as an exploit to take advantage of the vulnerability and attack unpatched systems (Packet Storm, 2020).

Even if vulnerabilities are not reported to the vendor or the public, attackers may discover them independently. Typically, as shown in the example above, patch development takes time and requires quality control. System owners are not always immediately aware of vulnerabilities or available patches for various reasons. For example, they are unaware of installed software or the usage of third-party components. Therefore, it is hard for them to monitor vulnerabilities, updates, and patches for their different systems, versions, and configurations 24/7. In addition, some vendors do not provide information about vulnerabilities and updates at all, or only for customers with an active maintenance contract. Furthermore, system owners may also choose to postpone installing new patches out of fear of unwanted side effects, like production downtimes. Thus, even patched vulnerabilities remain a threat, if the respective patches are not installed (Poremba, 2020). Wang et al. (2017) have analyzed more than 100,000 industrial control systems with *Shodan* over three years. Only 50% were upgraded to a new version within 60 days after vulnerability disclosure. In addition, according to Schneier, security vulnerabilities are inevitable, and there will always be a window of exposure (Schneier, 2000). Thus, smart security solutions for resilient systems have to be considered from the very beginning.

In order to detect relevant security vulnerabilities, and in turn, to react to reported vulnerabilities appropriately, automation support is needed to reduce the manual effort required for these tasks. In this paper, we present a model-driven framework for developing and managing multi-modal architectures, modes, and mode switches. If a vulnerability is detected, modes are switched automatically to mitigate and reduce the risk until software vendors provide patches, and system administrators install them.

In the context of our framework, we use modes to divide complex interconnected systems into logical, controllable, and tangible modes of operation (Hang and Hansson, 2013). Modes can have different purposes, as well as unique behavior. For example, airplanes have modes for parking, taxiing, take-off, manual and automatic cruising flight, and landing. A set of functionalities and configurations characterizes such modes. They can contain a plethora of different actions to be executed at various levels of abstraction, as modes are highly dependent on the domain and system they represent. In addition, some actions can be prohibited, like thrust reversal in the take-off mode of the airplane to prevent a crash. Therefore, we define a mode as a system state where a specific configuration is active and specific functionality is provided for a period of time.

Furthermore, the *Model-driven Engineering* (MDE) paradigm (Brambilla et al., 2017) provides tools, processes, methods for abstractly describing systems and components and automation support, e.g., via model transformations, to generate code for a specific instance automatically. Therefore, the goal of our work is to leverage MDE techniques to provide means for describing system configurations, modes, and mode switches, and to actively react to newly reported vulnerabilities.

We propose a *mode specification domain-specific language* (MDSL) to describe different components of a system, system modes, and specific actions to be performed for mode switching. Our DSL allows system administrators to specify their system definition and potential mitigation in case a vulnerability is detected. The DSL is embedded in our multi-modal framework that generates code fragments from the system definition. When the framework detects a vulnerability, a *Mode Control* component executes these code fragments to switch between different configurations or versions. These switches make a system more flexible (Yin and Hansson, 2018) in responding to external events and reducing the attack surface and the attackers' range of activity.

We, therefore, claim the following contributions:

1. An architecture and framework for switching system modes, e.g., when a vulnerability is detected;
2. A mode specification domain-specific language defining these modes and actions;
3. An evaluation of our approach by applying it to a web server case study and analyzing historical vulnerability reports. We further discuss the extent to which mode switching can reduce vulnerability times and attack threats.

The remainder of this paper is structured as follows. In Section 2, we present a motivating example. Related work on mode switching, the use of vulnerability scores, and applications of MDE for security concerns are presented in Section 3. In Section 4, we present our multi-modal architecture together with the mode-supporting domain-specific language and describe our prototype implementation of the framework in Section 5. We then present the results of a case study applying our DSL and mode switching for different web servers in Section 6. Finally, in Sections 7 and 8, we discuss the evaluation results and conclude with future work. We provide all data related to the study, the MDSL, and the mode-switching framework as open science material on Zenodo (Riegler et al., 2023): https://doi.org/10.5281/zenodo.7603904

2. Background and motivating example

We provide a brief motivating example from the domain of *Database Management Systems* (DBMS) on how mode switching can significantly reduce a software system's exposure to vulnerabilities and how automation support can ease the task of switching from one mode to another. Typically, applications use only one DBMS type, such as *MySQL* or the *Oracle Database*. If vulnerabilities are detected and subsequently reported, e.g., to the CVE database, administrators have to become aware of these vulnerabilities. Based on this information, they can determine whether their systems are affected. Specific vulnerabilities may only affect certain versions of systems or specific components thereof. Administrators have to either wait for the vendors and package maintainers to provide fixes, perform manual changes, update configurations, or roll back to a previous version at their own risk. It usually takes days and even months before fixes become available.

The same scenario holds for operating systems, web servers, and other software components. For example, Oracle typically provides critical patch updates only four times a year (Oracle, 2022), making their software more manageable for administrators, but sometimes more vulnerable. Things get worse when a vulnerability like *Log4Shell* (Hiesgen et al., 2022) affects multiple versions of software components, e.g., web servers in combination with databases. When attackers already exploit vulnerabilities, it is beneficial to have short-term protection ready at hand. We envision support for quickly switching to an alternative, less susceptible mode, without manually changing configurations, component versions, or services. Such an alternative can be, for example, a previous version of a component, another configuration, another container, or even a completely different application

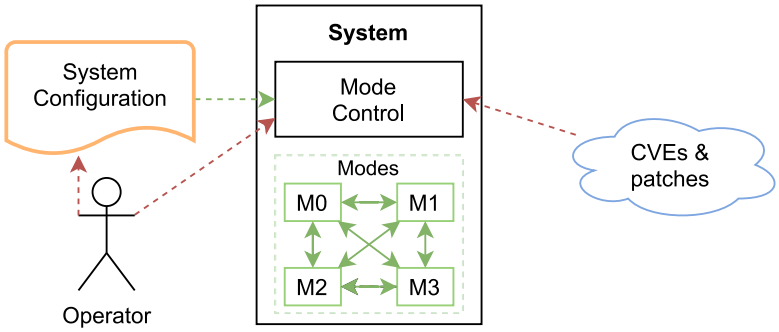


Fig. 1. Schematic overview of the envisioned mode-switching architecture.

that provides similar functionality. Doing manual adaptations is tedious and time-consuming, especially if several systems are affected. It is also prone to errors if applied hastily to mitigate potential threats. Therefore, automation support is highly desirable. Since multiple vulnerabilities may exist simultaneously, the ultimate goal is to select the alternative exhibiting the lowest risk. Depending on the mode switch's effect, specific changes to a previously active mode may be necessary once a vulnerability has been resolved. For example, if patches are eventually released and applied, returning from a degraded mode to the full-functionality mode has to be easy and needs to be performed in a controlled manner.

With our approach presented in this paper, we aim to describe modes on an abstract level with a supporting framework. Fig. 1 provides a high-level schematic overview of our envisioned architecture with four example modes M0 to M3. To realize this architecture, we are facing several challenges. Operators should be enabled to describe the modes a system can have, the modules that run in specific modes, the configurations, and the actions that need to be taken to stop or start a specific mode. We need a central *Mode Control*, which controls the system, uses the system configuration, and switches among predefined modes according to the risk level. To analyze the risk, Pedroza and Mockly (2020) use abstract weaknesses and attack patterns. In contrast, we consider concrete vulnerabilities from public databases. In addition, operators must be able to manually enable or disable modes if needed, e.g., for maintenance or in case of an emergency.

3. Related work

We have extensively studied modes and mode switches and have provided first findings of a systematic literature review on mode switching from a security perspective (Riegler and Sametinger, 2020). Our findings indicate that modes , in general, and mode switching, in particular, are widely used in various domains for different reasons, and on different levels. For example, modes are used to provide real-time adaptation, manage complexity, and adjust behavior.

3.1. Mode switching

Operating Systems (OS), provide a way to switch to a safe or protected mode with restricted access to certain OS functionality. Web browsers offer a compatibility or legacy mode and a private mode. Software applications offer a GUI, a terminal, and a debug mode with different features and functionalities. In addition, smartphones can typically activate an airplane mode, restricting access to wireless services, and user interfaces can be changed between the light and the dark mode. In nuclear power plants, different modes are used to combine configurations and to reduce complexity, e.g., for a safe shutdown in case of emergency (NUREG, 1995). The spacecraft, which took NASA's Perseverance Rover to Mars, switched to safe mode and turned off non-essential systems before landing because the measured temperature was outside the predefined range (NASA, 2020).

Another example is the standardized *Automotive Open System Architecture* (AUTOSAR) for electronic control units, which uses modes for overall states (Autosar, 2017). The vehicle itself, and each application and control unit part of a vehicle, can have different modes, influencing each other. For example, a low power mode for electric cars may reduce the power consumption of the air conditioner and inform the driver about low battery. Multi-function aircraft radars provide modes for air-to-air and air-to-ground operation, as well as several specific sub-modes (Hendrix, 2008). Modes are used for resource allocation, and task schedulability, as well as in the context of safety (Real and Crespo, 2004; Shih et al., 2018; Phan and Lee, 2011; Abeni and Buttazzo, 2001; Chen and Phan, 2018), which can be represented by finite state machines. Real and Crespo (2004) provide an overview of different protocols on how to change from one mode to another.

Modes further play an important role in system resilience and systems security engineering (Firesmith, 2019; Ross et al., 2016; Ross, 2014). As a reaction to errors, failures, or attacks, systems switch gracefully to a degraded or safe mode, but can still provide a minimum required amount of service (Glass et al., 2009; Knight and Strunk, 2004). For example, Avgeriou (2006) proposes runtime reconfiguration to satisfy *Quality of Service* (QoS). According to the principle of secure failure and recovery (Ross et al., 2016), systems should reconfigure themselves to ensure security policies.

In the context of *Software Product Lines* (SPL), we can compare the proposed mode-switching framework to reconfigure a deployed product configuration at runtime dynamically (Lee and Kang, 2006; Gomaa and Hussein, 2004; Göttmann et al., 2020). In addition to a single static configuration at design time, mode switching should provide flexible reconfiguration during the lifetime of a product or system to overcome vulnerabilities and attacks.

3.2. Moving target defense

An interesting related concept in our context is *Moving Target Defense* (MTD). This military strategy constantly changes systems, components, and configurations, making it more difficult for attackers to elaborate system information. As a kind of software behavior encryption, MTD is used to build resilient and fault-tolerant cloud services (Dsouza et al., 2013). Similarly, software diversification mitigates *break once, break everywhere* (BOBE) attacks on monocultures and increases fault-tolerance (Allier et al., 2015; Baudry and Monperrus, 2015). We have analyzed how

mode switching can improve the security of the intentionally insecure *OWASP Juice Shop*, see Riegler and Sametinger (2021a). Another approach by Thompson et al. (2016) rotates web servers to increase uncertainty and resilience. In contrast to their approach, our multi-modal architecture does not permanently change its mode, but only responds to specific events, e.g., when attacks are detected or when vulnerabilities or exploits become known.

3.3. Self-adaptive systems

Self-adaptive systems focus on configuring, reconfiguring, optimizing, and updating themselves automatically at runtime (Lemos et al., 2013). One prevalent reference model in this context is the *MAPE-K feedback loop* (Arcaini et al., 2015; Kephart and Chess, 2003) for managing and controlling autonomous and self-adaptive systems in various different domains, such as autonomous vehicles, IoT applications, or service-based systems. Monitoring, analysis, and subsequent adaptation play a fundamental role in these types of systems focusing on architecture adaptation, performance, or response time optimization (Moghadam et al., 2018; Khakpour et al., 2019; Barna et al., 2017), as well as ensuring reliability, for example, in robotic and cyber–physical systems (Cheng et al., 2020; Vogel-Heuser et al., 2015; Pradhan et al., 2016; Mosser et al., 2012; Gerostathopoulos et al., 2019). Some approaches in this area have also specifically addressed security concerns. For example, Tomić et al. (2018) present an approach for detecting attacks on network communication. Their approach operates on the network level and facilitates data routing adaptations. Abie et al. (2010) describe a messaging infrastructure facilitating adaptation and targeting security vulnerabilities with the main goal of improving reliability and robustness. However, their focus is on message-oriented middleware. In the domain of threat modeling, Van Landuyt et al. (2021) propose runtime support via reflective threat modeling to support adaptive security. While these approaches facilitate runtime adaptation and – to some extent – incorporate security concerns, they do not provide a flexible mode concept and a DSL as proposed in our mode-switching architecture.

3.4. Domain-Specific Languages (DSLs)

DSLs have been used to describe system components or adaptation strategies on various occasions. Again, in the domain of self-adaptive systems, Cheng and Garlan (2012) present *Stitch*, a language for architecture-based self-adaptation that focuses on repair strategies and related business objectives. Similarly, Arcaini et al. (2018) and Mirandola et al. (2019) propose the *MAPE* Specification Language, a DSL for adaptation patterns as well as cost-benefit considerations. While these approaches also allow changing the system at runtime, security aspects are the primary concern with our work. In security-related work, DSLs have also been successfully used. They were already used to define constraints for *Intrusion Detection Systems* (IDS) (Salgueiro and Abreu, 2010) or in the context of SPL (Beek et al., 2020). Additionally, DSLs were able to handle multiple security requirements in one system (Nguyen et al., 2015, 2017). In our context, the goal is to simplify the mode description when using our mode-switching framework.

4. Multi-modal architecture and mode DSL

In general, modes are highly domain-specific and can be implemented at various levels of abstraction. For example, a complete software system can have internal modes, describing its different states of operation. However, we can also describe modes as a set of components operating within this system. In the latter case, we can switch to a different mode by exchanging or modifying software components or modules that are part of the system. For example, we can exchange a concrete implementation of a web server, providing certain functionality, with a different implementation in many circumstances. While the implementation details and software components differ, the basic functionality is equal or similar enough. Therefore, the application remains operational regardless of the used implementation, i.e., the mode.

These different modes, in turn, serve as a means to reduce potential risks and maintain (essential) functionality during deployment when parts of the system become compromised by a detected vulnerability. In addition, modes further allow isolating critical functionalities into "fail-safe" modes, i.e., a minimal set of required operational components, from other functionalities. This principle has been successfully applied to medical devices, e.g., to apply different modes when malware was suspected (Sametinger et al., 2015). Similarly, the trustworthy multi-modal framework (Rao et al., 2018b) involves the incorporation of a composite risk model into the multi-modal design at every hierarchical level. The attack surfaces, vulnerabilities, and potential exploits of the modes are different. This difference can be utilized to choose the lesser risky option. Thus, we can use a multi-modal architecture to mitigate vulnerabilities, ward off attacks, and provide business continuity.

Based on these concepts, we have created a multi-modal architecture, augmented by a domain-specific language that allows defining modes at different abstraction levels. In the context of this paper, we focus on creating automated support in the form of a DSL and respective code generation to define different modes and their associated services at design time. This allows us to switch modes by deactivating a software module and starting another one as a replacement during runtime. When we perform such mode switches, the aim is to put the whole system at a lower risk. We consider only known risk that is documented in public vulnerability databases. These databases indicate the vulnerabilities and their severity scores known at a given time for the running system configuration.

4.1. System overview

Fig. 2 provides an overview of our proposed multi-modal architecture. The two main parts of the architecture are the *configuration part* (blue box in Fig. 2) with our domain-specific language and the *control part* (orange box in Fig. 2), responsible for analyzing vulnerabilities and performing the actual mode switches as defined in the system description. In the first step, we (1) define the desired modes using our domain-specific language. We refer to this definition as a *System Description*, as it contains simplified information about the different parts of the system, including actions to be performed to switch from one mode to another. Based on the system description, we automatically generate a *System Mode Configuration* (2), which contains system-specific information about all the software modules that run in various modes. Software modules are then identified by their *Common Platform Enumeration* (CPE) (MITRE, 2022a).

While the system is operational, the *Mode Control* component is responsible for collecting, analyzing, and reacting to relevant security vulnerabilities. As part of this, the *Vulnerability Manager* (3) automatically collects information regarding potential risks of vulnerabilities and exposures from the CVE database (MITRE, 2022b) of the modules as specified in our system definition. The same applies to patches collected from operating system repositories. In case the *Vulnerability Manager* identifies new vulnerabilities, relevant patches, or changed figures like the severity, (4) the respective information is forwarded to our *Analyzer*. Each CVE entry has a severity based on the *Common Vulnerability*

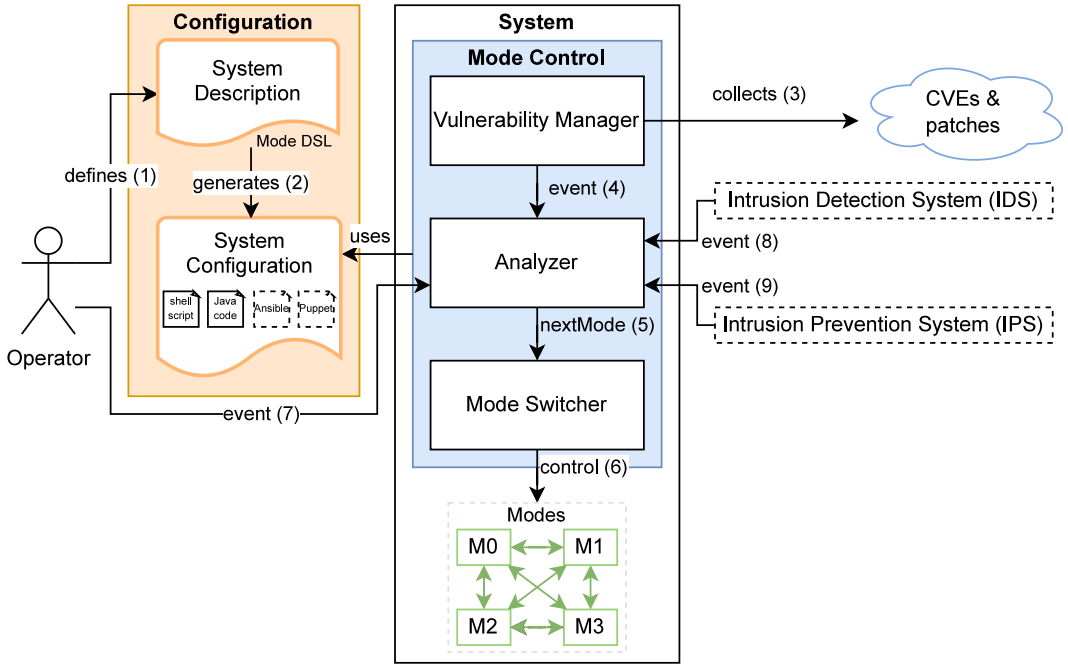


Fig. 2. Conceptual overview of our multi-modal architecture and workflow. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Scoring System (CVSS) (FIRST, 2022). This numerical score, ranging from 0 to 10, can be represented as low, medium, high, and critical, and is used to prioritize vulnerability management and remediation. The score depends on several metrics that consider the exploitability, the impact, and the environment. For each software and mode, the scores of unpatched vulnerabilities are added up. Based on vulnerability scores of all modules, (5) the *Analyzer* decides whether measures need to be taken, or whether the current mode of operation is still the most suitable one. If a mode switch needs to be performed (6), the new modules are selected and the *Mode Switcher* stops and starts services and executes the actions, e.g., shell commands, defined in the system description associated with the specific modules. In addition to automated mode switches based on reported vulnerabilities, we can trigger mode switches manually (7), for example, when we know that a specific module is at risk, or to precociously enable or disables specific modes to reduce the attack surface.

The architecture part with the mode-switch trigger is exchangeable and expandable, e.g., by intrusion detection systems (8) or intrusion prevention systems like *Fail2ban* (9). In addition to CVE entries, a vulnerability scanner, an IDS, runtime threat detection (Rao et al., 2018a; Hili et al., 2020) or a log file analyzer can also activate the *Analyzer* and ultimately lead to a mode switch.

4.2. MDSL – A domain-specific language for mode switching

In order to specify modes in a way that they are easily modifiable and reusable, e.g., by system administrators for external modes, or software engineers in case of internal modes, we have created a *Mode Domain-Specific Language* (MDSL).

Manually activating and deactivating components or creating multiple scripts executed for a mode switch is typically cumbersome and prone to errors, especially when multiple versions or component configurations are involved. We ease this task with a declarative specification of components, modules, and actions. Such a specification further eases subsequent maintenance of the components and their associated models. Combined with model-to-code transformation, we can automatically generate scripts or executable code based on the defined modes and modules. This facilitates the adoption of new components into the system and the definition of additional modes or updated versions of already existing modules without manually updating any code parts or scripts.

Fig. 3 depicts an overview of the meta-model of our MDSL, consisting of five main elements: the *System* that is modeled, constituent *Software* components, the *Modes*, the *Services* as well as the *Actions*.

The *System* element represents the system for which we define modes. This can be, for example, an operating system or web server consisting of multiple different parts, i.e., software components.

Software components serve as the building blocks that enable switching between different modes. Examples are database management systems, libraries, or specific applications. They can be activated/deactivated, i.e., started or stopped, by executing appropriate code or scripts. We describe each software by its name, vendor, product, and version. This information later serves as input for the automated CPE generation (cf. Section 4.3). Furthermore, we need the package name to identify software patches of the distribution's package maintainers. The package name highly depends on the naming convention of the distribution. For example, the package name for the Apache HTTP Server in Red Hat is `httpd` and in Debian it is `apache2`.

Mode: Once the software components are defined, we can specify the different modes, their different combinations and configurations of components. A mode represents a specific system configuration, running specific software components. We define each mode via its unique name, a textual description, a priority, and whether it is enabled. For example, "disabled" modes can be used for maintenance or recovery reasons. We use priorities to select a mode when two (or more) potential candidate modes are available. A mode uses a specific software set and relates to specific start and stop services executed at the mode switch. In order to support the ideas of moving target defense, specified alternative modes can be switched regularly.

Service: We use services to combine several actions in the correct sequence needed to start or stop specific software, e.g., a

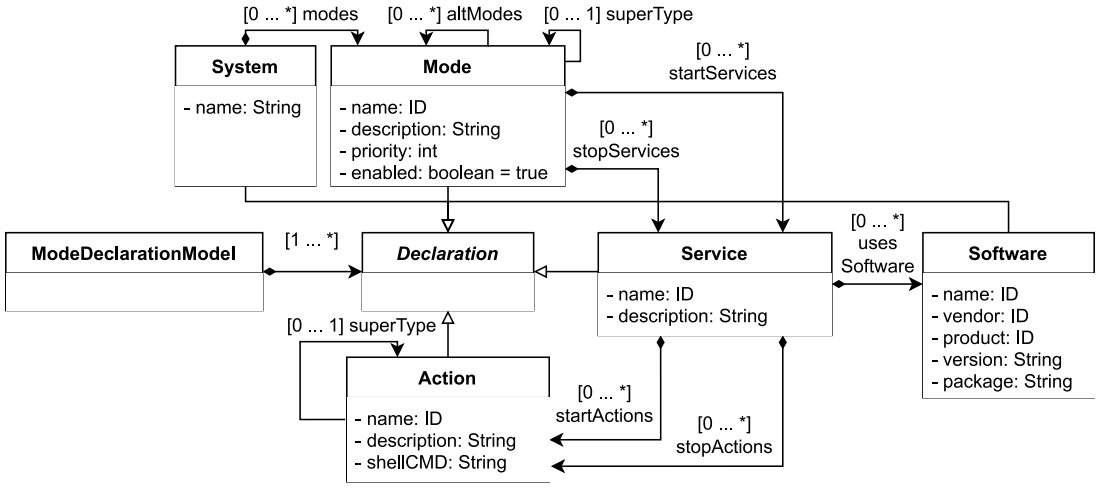


Fig. 3. Mode DSL meta-model.

database server. Services are executed when the receptive mode they are associated with is activated.

Action: In the simplest form, an action is a simple shell command executed to activate or deactivate a software component or service, e.g., the command `& ''C:\script.ps1''` to run a *PowerShell* script on Windows. In order to provide the possibility to define more complex actions, we can also use additional parameters when defining an action. We have also introduced inheritance to MDSL. Thus, one action can extend another one. Similarly, we can define a mode hierarchy that allows us to execute start and stop actions of the super mode before actions of its sub-mode. For example, the mode *Emergency* can have several sub-modes which become active in critical situations.

4.3. System modes and mode control

The system modes defined by our MDSL comprise specific services and start and stop actions that need to be executed when a mode is activated. For instance, these actions can use shell commands, e.g., to start or stop different services, execute programs, adapt configuration files to reflect the respective mode, or change specific properties and settings within an application. The multi-modal architecture builds on a set of components and modes that facilitates the use of common off-the-shelf software and defines security modes around it.

Mode Control, cf. Fig. 2, can either be directly integrated into a system, in case internal modes need to be handled, or it sits on top of the system to act as a "supervisor" and execute mode switches when necessary. The *Vulnerability Manager* uses the *System Configuration* generated from the MDSL to identify relevant software modules (as specified and used in the different modes). This information is used to build CPEs and to periodically retrieve CVEs from various online resources, such as the NVD (NIST, 2022a,b). In case relevant information is fetched from the CVE resource, this information is passed on to the *Analyzer* component. Information is relevant when a new vulnerability has been reported, when a patch has become available, or when vulnerability scores have changed. Based on this information, the *Analyzer* suggests the least risky mode. The priority, which we define in MDSL, is considered to choose among modes with identical vulnerability scores. For example, we can use priorities to specify that we prefer to run a certain software module and only run the alternative if its risk is lower. If mode rotating (Thompson et al., 2016) is enabled, and one of the alternative modes has the same risk as the current mode, the alternative is suggested. The *Mode Switcher* will execute the mode switch suggested by the *Analyzer*. It will stop the current mode and start the new mode with predefined services and actions. If the suggested mode is already running, then the *Mode Switcher* has nothing to do, and the current mode remains active.

If certain services are used in two modes, then we keep them running when switching from one mode to the other. This partial stop/start reduces the possible downtime between a mode switch. Furthermore, the administrator can change the mode priorities and manually disable or enable modes as a precaution, e.g., when vulnerabilities or exploits become known but are not yet documented, or when something unforeseen happens.

5. Prototypical implementation

We have created a *Java* prototype implementation of our multi-modal framework to support experimentation and serve as a proof-of-concept for our mode-switching approach. This includes the domain-specific language for defining modes, services, actions, the mode control components, and the code generator. We briefly report on the details of the implementation, focusing on the model-based generation of the system configuration and *Mode Control*.

Listing 1 shows an excerpt of the MDSL grammar following the concepts and functionality outlined in Section 4.2. We have implemented our MDSL based on the *Eclipse Modeling Framework* (EMF) as a meta-model and provide a textual syntax using the *Xtext* framework (Eclipse Foundation, 2022). In order to ensure valid system configurations, we implemented a validation component that, for example, verifies package names according to the detected or specified operating system, to prevent duplicates and logical errors like self-inheritance. To ease the task of defining a system description, we provide several predefined default actions, such as starting and stopping specific services, without manually defining these commands for each operating system.

Once a system description has been defined or modified using our MDSL, the respective system configuration can be automatically created within the *Eclipse Editor* or by running the *Mode-Switching Framework* on the command line. We use *Xtend* (Eclipse Foundation, 2020) to generate *Java* code from the file with the extension `.mdsl` to define modes and actions, which are executed on a mode switch. Then we create the *Java* bytecode from it. Listing 2 shows an excerpt of the *System Configuration Generator*. For the evaluation described in Section 6, we have further implemented a crawler to collect corresponding CVEs and patches, which can then, in turn, trigger mode switches once a vulnerability is reported.

```
Model: (declarations+=Declaration)*;
Declaration: System | Software | Mode | Service |
    Action;
System: 'System' name=ID 'has' 'following' 'modes'
 (modes+=[Mode] (',' modes+=[Mode])*)?;
Software: 'Software' name=ID 'identified' 'by'
 'cpe:/:'vendor=ID':'product=ID':'version+=STRING
 'package' package=STRING;
Mode:
 'Mode' name=ID ('extends' superType=[Mode])?
 'description' description=STRING 'priority'
     priority=INT
 'startServices' (startServices+=[Service]
    (',' startServices+=[Service])*)?
 'stopServices' (stopServices+=[Service]
    (',' stopServices+=[Service])*)?
 (enabled=Enabled)?;
Service:
 'Service' name=ID
 ('usesSoftware' usesSoftware+=[Software]
    (',' usesSoftware+=[Software])*)?
 'startActions' startActions+=ActionWithParams
    (',' startActions+=ActionWithParams)*
 'stopActions' stopActions+=ActionWithParams
    (',' stopActions+=ActionWithParams)*;
enum Enabled: true | false;
ActionWithParams: action=[Action]'('(params+=STRING
     (',' params+=STRING)*)?')';
Action:
 'Action' name=ID ('extends' superAction=[Action])?
 ('description' description=STRING)?
 'shellCmd' shellCmd=STRING;
```

Listing 1: Mode Domain Specific Language (MDSL)

```
public static SystemConfiguration create() {
 List<Mode> modes  = new ArrayList<Mode>();
 ...
 <<FOR m : model.declarations.filter(Mode)>>
  Mode <<m.name.toFirstLower>> = new Mode("<<m.name
      .toFirstLower>>","<<m.description>>", <<m.
      priority>>,
   Arrays.asList(
    <<IF m.superMode !== null>>
     <<m.superMode.startServices.map[name.
         toFirstLower]
     .join(', ')>>
     <<IF m.startServices !== null>>,<<ENDIF>>
    <<ENDIF>>
    <<m.startServices.map[name.toFirstLower].join('
        , ')>>
   ),...)
  );
 <<ENDFOR>>
 ...
 return new SystemConfiguration(modes, software);
}
```

Listing 2: Excerpts of the Xtend System Configuration Generator

6. Case study: Web server modes

To demonstrate the expressiveness, succinctness, performance, and potential benefits of our approach, we have conducted a case study using different web servers. Following the guidelines described by Runeson and Höst (2008), we have replayed the period of two years from Feb. 2019 to Feb. 2021. We explore the following **Research Questions (RQs):**

RQ1: How expressive and succinct is our MDSL for describing a real-world system configuration, compared to manually defining the modes on the code level? With this question, we investigate to what extent our DSL can be used to define the different components, the different modes, and the different actions that are required to switch from one mode to another.

RQ2: How fast is our multi-modal framework in processing the system instance configuration, fetching vulnerabilities and patches, comparing mode risks, and executing mode switching? With this question, we assess the performance of our architecture and its implementation to collect and analyze vulnerabilities and perform mode switches accordingly.

RQ3: To what extent can our switching approach reduce the risk of a system being vulnerable and increase its resilience? Finally, with this question, we assess the potential benefits of applying mode switching in the case study context by collecting and analyzing reported vulnerabilities for this type of system recorded in the past.

Requirements: In order to accurately assess the expressiveness, succinctness, and performance of our framework and DSL, we need a system that allows for several different modes and actions, to explore how well MDSL can capture the different modes, and how fast our framework processes the system instance configuration. Furthermore, the system, more specifically the used software, must have known and documented vulnerabilities and patches, which we can collect, calculate the CVSS sum for each mode, and subsequently, decide whether to execute a mode switch.

Selected Use Case: Based on these requirements, we have selected the two most commonly used web servers *Apache* and *Nginx*, combined with the *PHP: Hypertext Preprocessor* (PHP). We have selected this combination for our study because it is used by many web applications and by many common *Content Management Systems* (CMSs) such as *WordPress*, *Drupal*, or *Joomla* in the real world. Typically, an instance of an application uses only a single web server. If a vulnerability occurs, software vendors need to react quickly to provide an update. Then the system owner can test and install it, before attackers take advantage of the vulnerability of online systems. As the web servers and PHP were regularly affected by vulnerabilities in the past, there is a need for improvement. The use case fits the requirements as we can derive modes and have a long history of vulnerabilities and patches. Fig. 4 depicts the use case with the web servers, the files they can provide, and possible mode switches shown as dashed lines. The modes *Apache* and *Nginx* can provide only static web content such as HTML pages. In addition to that, the modes *ApacheWithPhp* and *NginxWithPhp* extend the other modes with PHP functionality and can provide dynamic content from a database.

Evaluation Settings: For our evaluation on a virtualized Linux distribution Debian 10 (Buster) with 2 GB RAM on an Intel Core i7-8565U machine with 16 GB RAM, we created a system configuration with two different implementations of two popular web servers in their most recent versions: (*i*) Apache2 (version 2.4.38) and (*ii*) nginx (v1.14.2). Additionally, running on the web server, PHP (v7.3) and FastCGI Process Manager are used to serve dynamic web content. Both web servers were selected because they provide similar functionality and work together with PHP. MariaDB (v10.3.29) was selected as a MySQL DBMS and WordPress 5.8 as a CMS. The web content was saved to the default `/var/www` directory, such that both web servers had access to it.

For RQ2 and RQ3, we have simulated and replayed our web server scenario from Feb. 2019 to Feb. 2021, based on the software components and versions described above. We used this period because there were no major software changes. Based on a CPE such as `apache:http_server:2.4.38`, we used NVD (NIST, 2022a) to retrieve relevant vulnerabilities (CVEs). NVD provides a CVE summary, the severity with CVSS version 3.1, some additional links to the vendor or third-party advisories, as well as an assignment to the *Common Weakness Enumeration* (CWE) (MITRE, 2022c) for each vulnerability. However, NVD only provides the

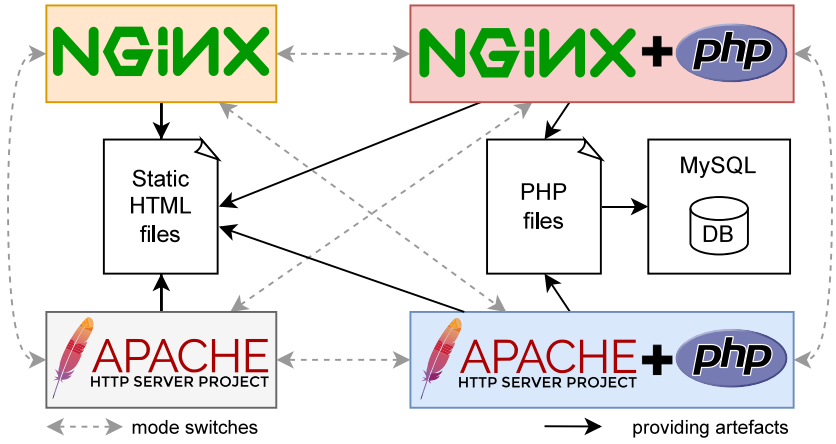


Fig. 4. Multi-mode web server scenario.

published date and the last modified date of a CVE, so we manually inspected vendor websites for initial report dates to security teams and update release dates to calculate the days until the update and the patch was provided. In addition, we have investigated when the CVEs were fixed in the corresponding packages according to the Debian security tracker. In stable releases, Debian does not offer all the newest software versions but aims at production stability (Debian, 2022b). The security team provides updates and advisories for about three years and backport actual fixes to older versions (Debian, 2022c).

Evaluation Metrics: We further refer to the results of each evaluation metric with its unique identifier M1.1–M3.3.

For RQ1, we focused on the expressiveness and succinctness of our approach to the selected case. We measured the *Lines of Code* (LoC) and the *Halstead Software Complexity Measures* (Halstead, 1977) of the MDSL system description (M1.1) and compared them with the generated system configuration (M1.2). These metrics provide a way to objectively evaluate the complexity and quality of the MDSL. Without the MDSL, a user would be required to develop the system configuration manually.

For RQ2, we measured the execution time at design-time and runtime. After defining the system description with the MDSL, we measured the time to validate the system description (M2.1), and how long it takes to generate (M2.2) and to compile (M2.3) the system configuration. At runtime, we measured the time to fetch vulnerabilities and patches (M2.4), to calculate the CVSS sum and compare the modes (M2.5), and finally, to switch the mode (M2.6). The mode-switching duration should be below the common HTTP timeouts from 30 to 120 seconds (Thomson et al., 2010). Otherwise, the user's browser displays an error message after the waiting time.

Based on real-world software vulnerabilities and patches, for RQ3, we calculated the CVSS sum (M3.1) and the risky days (M3.2) for each mode and day manually and automatically with our framework. Finally, we compared the results with our mode-switching approach (M3.3).

6.1. RQ1: MDSL expressiveness & succinctness

Research Design: For assessing the expressiveness and succinctness of our approach when defining components of a multi-mode system, we created a system description of the different software, modes, services, and actions. For this, we defined four modes for our system and have ordered them by their priority: *ApacheWithPhp*, *NginxWithPhp*, *Apache*, and *Nginx*. Listing 3 shows an excerpt of our MDSL web server system description. In the first step, one researcher created the different software components, each identified by a unique name, e.g., the corresponding Debian package name or the CPE. Afterward, the respective modes were defined. For instance, mode *ApacheWithPhp* extends the general mode *Apache* with additional actions to enable required Apache modules and configurations and to start the Linux service for the PHP FastCGI Process Manager. In the final step, we created the individual actions that are performed to switch to a new mode. For instance, the Linux-specific shell commands (`servicectl start`) to start a service.

```
System Webserver has following modes Apache,
    ApacheWithPhp,Nginx,NginxWithPhp
    operatingSystem Linux distribution Debian

Software apache2
Software nginx
Software php7 package "php7.3"

Mode Apache
description "Only static HTML pages" priority 3
startServices apache2Service stopServices
    apache2Service

Mode ApacheWithPhp extends Apache
description "Static HTML and dynamic PHP pages"
    priority 1
startServices apacheModProxyFcgiService,
    apacheModSetenvifService,phpFpmService,
    apacheConfPhpFpmService,apache2restart
stopServices apacheConfPhpFpmService,phpFpmService,
    apacheModSetenvifService,
    apacheModProxyFcgiService,apache2restart
...
Service apache2Service usesSoftware apache2
startActions startService("apache2")
stopActions stopService("apache2")
...
Action startService shellCmd "/usr/bin/systemctl
    start"
Action stopService shellCmd "/usr/bin/systemctl
    stop"
```

Listing 3: Excerpts of the MDSL Web Server System Description

After defining the system description with the MDSL, the Java-based system configuration was automatically generated and compiled. Then we calculated and compared their LoC and the *Software Measures* by Halstead (1977) in Table 1. In the following, we briefly explain the measures. The *Vocabulary* (n) helps to understand the complexity and represents the total number of

Table 1
Comparison of the Lines of Code (LoC) and the Halstead's Software Complexity Metrics of the MDSL System Description and the generated Java-based System Configuration.

Metric	LoC	Vocab.	Size	Volume	Difficulty	Effort
Java	95	75	566	3525.51	38.75	136613
MDSL	59	69	206	1258.36	29.25	36806
Reduction (rel)	38%	8%	64%	64%	25%	73%

unique operators ($n1$) and unique operands ($n2$) used in the code: $n = n1 + n2$. The *Size* (N) or program length is the total number of elements (operators [$N1$] and operands [$N2$]): $N = N1+N2$. The *Volume* (V) estimates the needed program space and is proportional to the size. It is calculated by the size multiplied with the 2-base logarithm of the vocabulary: $V = N * ld(n)$. The *Difficulty Level* (D) shows how difficult it is to handle the program. It is proportional to the number of unique operators and is calculated as $D = (n1/2)*(N2/n2)$. The *Effort* (E) is related to the mental activity needed to implement or understand the program and is calculated as the volume multiplied by the size: $E = D * V$.

Results: Describing a system using MDSL showed that only 59 LoC with 69 unique operators and operands vocabulary (M1.1) were needed to define software components, modes, and actions compared to 95 LoC and a vocabulary of 75 for the generated Java-based system configuration (M1.2). MDSL provided a reduction of 36 LoC, 6 tokens in the vocabulary, and 360 tokens in the program size. Modes like *ApacheWithPhp* build a logical frame around the web server and additional described services. No further additions or modifications in the used software code were necessary. Inheritance allowed us to extend the mode *Apache* with PHP services as mode *ApacheWithPhp*. This mechanism reduces redundancy, lowers size and volume, and increases maintainability because changes like adapting actions or software versions only have to be made in one place. For the given use case, our MDSL allows focusing on the arrangement and declarative definition of modes and shows a reduction of 25% for the difficulty level. In addition to the improvement of the evaluated metrics, we would like also to emphasize that MDSL provides more functionality than a simple text file. Tool support provided by the Eclipse framework, such as code completion, syntax highlighting, and our configuration validation component, eases the task of creating modes and actions, thus increasing usability. We statically analyze the system description and show error messages if there are duplicates of mode names, priorities, or action names. If the Debian OS and the distribution are specified, we check if the software and the packages are within the official package list. If not, we provide a respective warning message.

Answer to RQ1: We have shown that MDSL was expressive enough to describe the given case and the respective modes on an abstract level while also providing syntax highlighting, code completion, and validation by dedicated tool support. In addition, using MDSL leads to more succinct specifications compared to manual coding. The system description in MDSL shows a reduction of 38% in LoC, 64% in size and volume, and 73% in the expected testing efforts when compared to the generated Java-based system configuration.

6.2. RQ2: Performance of the multi-modal framework

In the second part of our case study, we used our general multi-modal framework and measured the performance at design-time and runtime.

Research Design: We analyzed the average runtime of 500 executions to validate the user-defined system description and to generate and compile the system configuration automatically. At runtime, we used our framework to fetch CVEs from NVD (NIST, 2022b) and patches from the Debian Security Tracker (Debian, 2022a) based on the specified software (vendor, product, and version) and measured the performance. Then, we replayed the appearance of CVEs and patches over the last two years. For each software, mode, and day, we compared the calculated vulnerability scores with manual calculations. We reviewed whether the correct next mode was suggested based on the lowest risk and highest priority. Additionally, we simulated and analyzed nearly 2500 mode switches. We examined the correct execution and overlaps between old and new modes, e.g., old mode services that are needed in the new mode continue to run, and only services that are no longer needed are stopped. Finally, we analyzed the mode-switching effects on currently running requests and logged-in users. Therefore, the upload of a 50MB file and the session handling at the WordPress CMS was studied.

Results: We were able to use the system description and the generated system configuration in our framework. The Xtext parser validates the system description in about 3.1 milliseconds (ms) (M2.1) if the syntax conforms to the defined MDSL. If successful, our framework automatically generated the Java system configuration in 1.3 ms, on average (M2.2). However, compilation took much longer with 204.2 ms (M2.3). After starting our framework, the mode with the lowest CVSS sum and the highest priority was selected. On average, we have fetched the CVEs in 416.5 ms and the patches in 251.8 ms (M2.4). However, this runtime highly depends on the Internet connection, server availability, and rate limits of the online repositories. NVD provided some outdated and already fixed vulnerabilities for our CPEs or did not consider the used operating system or distribution. To compensate for these inaccuracies, our framework ignores CVEs reported before the distribution's release year. Additionally, it happens that vulnerabilities were assigned to different packages or specific to Windows. For example, NVD assigned CVE-2019-9517 to CPE `apache:traffic_server` while the *Debian Security Tracker* used the package *apache2*. The other way round, NVD considered CVE-2019-19246 as PHP vulnerability, but Debian package maintainers assigned it to the PHP sub-package *libonig*. That makes it harder to link vulnerabilities and patches. To take that into account, the *Vulnerability Manager* considers CVEs from Debian Security Tracker with status resolved or urgency unimportant as fixed. In total, 29 of 74 CVEs from NVD were excluded because 23 were already resolved in the Debian distribution, three were vulnerabilities of other software parts, and three were specific to Windows.

The calculation of the CVSS sum for each mode and the comparison took 0.14 ms on average (M2.5). The next modes were automatically suggested, matching to our manual evaluation. The mode switches were correctly executed, and partial start and stop functionality was used. Table 2 shows the average duration of mode switches (M2.6). It takes about 10.3 ms to start the *Apache* mode, whereas it takes 180.5 ms to start *ApacheWithPhp* and even longer (191.5 ms) to switch from *Nginx* to *ApacheWithPhp* and back (185.6 ms). Except for longer running requests like uploads, it became obvious that switching between the two modes with PHP had no negative side effects on users currently logged in to WordPress because the session information on the server and the client were not changed and was saved in the database.

Answer to RQ2: At design-time, it takes 4.1 ms to validate the system description, 1.3 ms to generate the system configuration, and 204.2 ms to compile it. At runtime, it takes less than a second in total (761.9 ms), to collect vulnerabilities (416.5 ms) and patches (251.8 ms), to analyze the modes CVSS sum (0.14 ms), and to perform the necessary mode switches (93.5 ms). The collection and analysis was done in the background and did not affect current users. The mode switch takes from 10.3 to 191.5 ms, which is far below the required lower HTTP timeout of 30 s (30,000 ms).

Table 2
Average duration in milliseconds of 2500 mode switches.

From	To				
	Apache	*Apache WithPhp*	*Nginx*	*Nginx WithPhp*	*inactive*
Apache	–	176.9	64.7	116.3	13.8
ApacheWithPhp	121.6	–	185.6	163.7	141.4
Nginx	32.5	191.5	–	49.3	21.5
NginxWithPhp	48.1	148.8	16.7	–	36.2
inactive	10.3	180.5	51.7	98.2	–

Table 3
Overview of the different web server components, the number and severity of the vulnerabilities, and time until updates and patches were available.

Software	CVEs	CVSS 3.1		Days4Update		Days4Patch	
	#	*sum*	*avg*	*sum*	*avg*	*sum*	*avg*
apache 2.4.38	17	117.3	6.9	1318	77.5	467	27.5
nginx 1.14.2	4	26.8	6.7	284	71.0	46	11.5
php 7.3.5	24	169.3	7.1	1871	78.0	1712	71.3
Total	45	313.4	7.0	3473	77.2	2225	49.4

6.3. RQ3: Mitigating vulnerabilities by switching modes

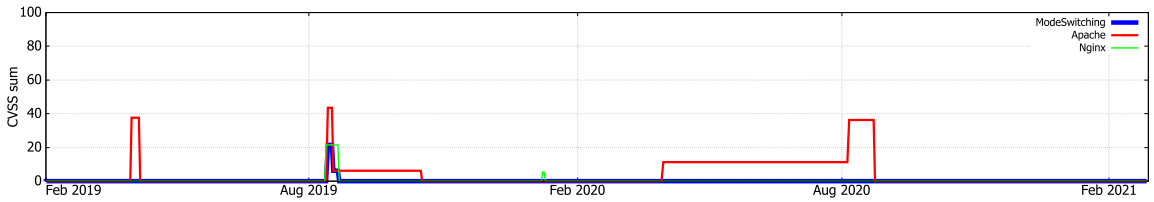
In the third part of our case study, we used the previously defined modes and the multi-modal framework to simulate a time span of two years with real vulnerabilities and patches in a web server scenario. The simulation has shown the effects of previously reported vulnerabilities and to which extent mode switching can lower the risk of being susceptible to potential attacks.

Research Design: In our sample web server scenario, we had four different modes. In order to reduce the risk of our web server, we switched modes whenever there was another mode that provided less risk. We simulated our web server scenario for a time span of two years, from February 2019 to February 2021, and calculated the risk of each mode for each day by retrieving information about unpatched vulnerabilities and their corresponding score. Whenever another mode had less risk (a lower score), we switched to this mode. The score was calculated again if a patch was provided, possibly leading to another mode switch.

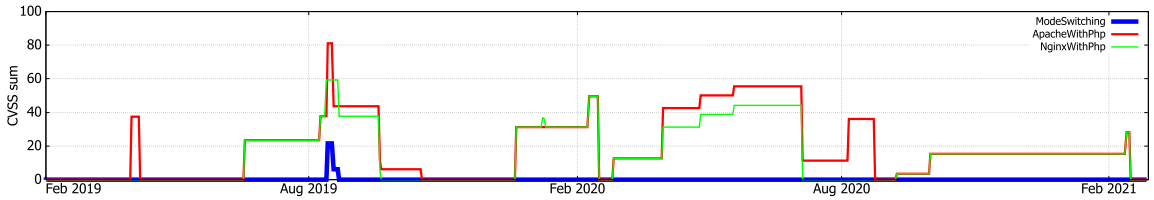
Table 3 shows the number of CVEs for the used software, the CVSS scores, and how long it took the vendor respectively package maintainers to provide updates (Days4Update) and patches (Days4Patch). Software vendors have provided updates for the 45 relevant vulnerabilities within an average of 77.2 days. This number is similar to the 77.5 days of vulnerability statistics by Edgescan (2019). The Google Project Zero (Cimpanu, 2019) reports that they resolve 95.8% of found security bugs before the 90-day deadline of common public disclosure. According to the statistics, the software nginx 1.14.2 had fewer vulnerabilities than Apache 2.4.38 and lower CVSS scores. PHP 7.3.5 had more CVEs than Apache and nginx combined, and the deployment of patches (Days4Patch) took 71.3 days on average. Overall, it took the Debian package maintainers 49.4 days on average to analyze vendor updates and to provide patches for the used software. After that, system owners were able to install the patches. Using this software resulted in a window of exposure of 126.6 days on average (Days4Update: 77.2 + Days4Patch: 49.4), from the report to the vendor's security team until the package maintainers provided the patch.

Out of 45 vulnerabilities, we found five publicly available exploits on *Packet Storm* (Offensive Security, 2022) and the *Exploit Database* (Packet Storm, 2020): four in Apache and one in PHP. On average, exploits were published 50 days after the vendor update and 32 days after the distribution patch. Three exploits were released very close to the patch date. However, there can be both unpublished vulnerabilities and unpublished exploits. In our sample scenario, we did not have public disclosures of vulnerabilities before vendor updates came out. Software vendors rarely publicly share reported new vulnerabilities or share information with customers before providing an update, so we only used the provided data by CVE sources. We have considered the time span between vulnerability publication and the patch date provided by the Debian package manager.

Results: Fig. 5(a) shows the results of using only *Apache* (red) or *Nginx* (green) compared with the mode-switching results (blue) for the analyzed time span of two years. We have calculated the total CVSS 3.1 score for each mode and each day. New vulnerabilities increased the total score, and corresponding published patches reduced it, assuming that patches were installed as soon as they were deployed. *Apache* (red) had several periods with a higher CVSS score in Apr. 2019, from Aug.-Oct. 2019, and from Apr.-Aug. 2020. The maximum CVSS score of 43.5 was on Aug. 14, 2019. *Nginx* (green) had only two short periods in Aug. 2019 and Jan. 2020 with a higher CVSS score and a maximum of 21.5. Our framework switched automatically to the mode with the lowest CVSS score and highest priority. Thereby we reduced the risk on 751 of 759 days (98.9% of the time) to a zero CVSS score. Only eight days from Aug. 14–21, 2019 had a higher CVSS score (max 21.5), because both *Apache* and *Nginx* had unpatched vulnerabilities. On Aug. 14, 2019, there was a mode switch from *Apache* (43.5) to *Nginx* (21.5). Because some *Apache* vulnerabilities were fixed and the CVSS score was reduced to 6.1 on Aug. 18, 2019, our framework switched back to *Apache*. Without PHP the web servers *Nginx* and *Apache* can only provide static pages. Using PHP allows generating dynamic content with template systems, CMSs, or frameworks. Nevertheless, that also involves greater risks, as we see in the results of the PHP modes *ApacheWithPhp* (red) and *NginxWithPhp* (green) in Fig. 5(b). We extended the analysis and compared them with mode switching (blue). PHP and its vulnerabilities (see Table 3) increased the number of risky days of the PHP modes on 309 of 759 days (40.7% of the time). Table 4 shows a comparison of the risky days and the CVSS score of the modes. Without mode switching and using only one mode, we have 11 (*Nginx*) to 536 (*ApacheWithPhp*) risky days (M3.2) and a maximum total CVSS score of from 21.5 (Nginx) to 81.2 (ApacheWithPhp). Using *ApacheWithPhp* 70.6% of the analyzed time was risky, with an average total CVSS score of 18.5 (M3.1). Compared to that, mode switching between all modes resulted in only 1.1% risky days and an average total CVSS score of 0.1. Switching only between the PHP modes *ApacheWithPhp* and *NginxWithPhp* did not reduce the risky days in the analyzed time span, because most of the vulnerabilities were caused by PHP. Using *NginxWithPhp* provides the same amount of 442 risky days (58.2% of the time). However, mode switching between the PHP modes leads to a lower CVSS sum (11255.4) and lower average CVSS (14.8) instead of using only the mode *ApacheWithPhp* (14027.5, 18.5) or *NginxWithPhp* (11349.1, 15.0). At the critical PHP vulnerability CVE-2019-11043 with a CVSS of 9.8, mode switching was tight, because the vendor update, the CVE, and the fix in the distribution were published on the same day as the exploit (Lerner, 2019). Meanwhile, there is already a *Metasploit* module available, which makes it even easier to take advantage of the vulnerability. Our approach helped in 4 out of 5 (80%) of the mentioned exploits to reduce the attack surface and helped to increase resilience in such cases where it was close to fixing a problem before it became publicly known. Compared to the modes with the maximum and minimum risky days, mode switching between all modes has 3 (27%) to 528 (98%) fewer risky days than using only one mode (M3.3).



a) CVSS Sum Mode-Switching (blue), Apache (red) and Nginx (green)



b) CVSS Sum Mode-Switching (blue), ApacheWithPhp (red) and NginxWithPhp (green)

Fig. 5. Total vulnerability scores. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 4
Comparison of the modes with and without PHP, and mode switching: the number and percentage of risky days compared to the total time, and the total, average and maximum severity scores of the vulnerabilities.

Mode	Risky days		CVSS 3.1		
	#	%	*sum*	*avg*	*max*
ApacheWithPhp	536	70.6	14 027.5	18.5	81.2
NginxWithPhp	442	58.2	11 349.1	15.0	59.2
Apache	217	28.6	2 882.5	3.8	43.5
Nginx	11	1.4	204.1	0.3	21.5
Mode switching (All modes)	8	1.1	110.4	0.1	21.5
Mode switching (PHP modes)	442	58.2	11 255.4	14.8	59.2

Table 5 shows mode switching simulations with different priorities and different available modes, which resulted in different active days for each mode and a different number of mode switches (#). If a new vulnerability or patch occurred, modes were switched depending on the total CVSS scores and the priorities. If the web server Apache had a higher priority as shown in Table 5a, in total 17 modes switches were needed and the modes *ApacheWithPhp* and *Apache* were running 70% (29.4 + 41.2) of the time. If we preferred Nginx as shown in Table 5b, in total 14 modes switches were needed and the modes *NginxWithPhp* and *Nginx* ran even 99.1% (41.8 + 57.3) of the time. Because CMSs like WordPress need PHP, using only the corresponding modes with PHP are shown in Table 5c and d. Preferring *ApacheWithPhp* led to 11 mode switches in total and a nearly equal amount of active days (*ApacheWithPhp* 58.9%, *NginxWithPhp* 41.1%). If *NginxWithPhp* had a higher priority, it was running 99.1% of the time. However, seven mode switches were still executed, and *ApacheWithPhp* was active for seven days, because this mode had a lower CVSS score.

Answer to RQ3: Mode switching reduced the risk to a zero CVSS score in 98.9% of the time. Only eight days (1.1% of the time) had the lowest CVSS score at 6.1, respectively 21.5, because all the software used in the modes had unpatched vulnerabilities. If PHP is needed for specific CMSs, switching modes only between the PHP modes showed no improvement in the number of risky days. However, the results provided a lower total CVSS sum (−93.7) and a lower average total CVSS (−0.2) compared to using the mode *NginxWithPhp* only.

6.4. Threats to validity

Like any other study or experiment, our work is subject to threats to validity (Wohlin et al., 2012). Concerning *conclusion validity*, we calculated and compared the total vulnerability scores for each day, both with and without mode switching. Regarding our evaluation in this study, we have demonstrated that switching between different modes in the context of web server vulnerabilities can lower the risk of being susceptible to attacks. To further evaluate our approach, we plan to conduct a usability study with engineers from industry to analyze MDSL and the concept of our multi-modal architecture in an industrial context.

Construct validity is concerned with the relationship between theory and observation. Prioritizing remediation efforts with CVSS is an open industry standard and well established. Currently, we are only considering the severity of the vulnerabilities. As an extension to this metric, we envision additional analysis performed by our architecture, e.g., taking into account how long a vulnerability is open, how often vulnerabilities occur for specific components, and whether there are exploits. We rely on the fact that our modes for *Apache* and *Nginx* (as well as *ApacheWithPhp* and *NginxWithPhp*) are functional and non-functional mostly equivalent. However, the modes without PHP provide more security, but less functionality. As part of our ongoing work, we are currently working on adding pre and post-conditions to the mode-switching rules to check, if the new suggested mode is adequate for the given scenario, e.g., does the mode provide specific features, and if the mode switch was carried out successfully. In our MDSL evaluation, we considered the lines of code and *Halstead Software Metrics* to measure software complexity and quality objectively. However, we plan to analyze additional factors, such as maintainability and portability.

Regarding *internal validity*, the case study was examined in a virtual environment with historical CVE and patch data, such that user traffic, changed CVE entries, or changed patches did not influence the mode-switching evaluation. In order to mitigate other factors on the duration measurements, we performed 2500 mode switches and averaged them.

Concerning *external validity*, we used common open-source software to present our multi-modal approach. Millions of internet domains widely used the analyzed web servers, which had several CVEs and patches over time. Therefore, we are confident that developers can use our DSL and framework and adapt it to their specific systems and requirements, having similar characteristics to the shown case study. However, we cannot claim

Table 5
Simulations with different mode priorities and available modes: modes ordered by priority ascending, number of active days, percentage of active days (%), number of switches to the mode (#).

(a) All modes - Priority Apache

Mode	*days*	%	#
ApacheWithPhp	223	29.4	5
NginxWithPhp	94	12.4	4
Apache	313	41.2	5
Nginx	129	17.0	3

(b) All modes - Priority Nginx

Mode	*days*	%	#
NginxWithPhp	317	41.8	5
ApacheWithPhp	0	0.0	0
Nginx	435	57.3	7
Apache	7	0.9	2

(c) PHP modes - Priority Apache

Mode	*days*	%	#
ApacheWithPhp	447	58.9	6
NginxWithPhp	312	41.1	5

(d) PHP modes - Priority Nginx

Mode	*days*	%	#
NginxWithPhp	752	99.1	4
ApacheWithPhp	7	0.9	3

that our framework applies to any given environment or technology, showing different characteristics in terms of generalizability. A user study may be necessary to validate the feasibility of our framework. Nevertheless, we think the general multi-modal architecture can be applied to different domains, as long as information regarding vulnerabilities and alternative modes are available.

For *reproducibility*, we provide a reproduction package for the study on Zendodo (Riegler et al., 2023). It is possible to reuse the system definition and configuration with our executable Java jar-file and to simulate/replay the two years of the case study to see the mode switching and the reaction to CVEs as well as patches. There might be slight changes in the results, due to possibly changed severities of CVEs in the meantime. For example, this can happen, when new exploits become available for existing vulnerabilities.

7. Discussion

Our evaluation has confirmed that our multi-modal architecture with the accompanying MDSL is capable of describing different system components, modes, and actions to be executed. In the following, we further discuss findings, practical implications, as well as current limitations of our work in a more general way.

7.1. A domain-specific language for mode switching

We used the presented scenario to study the benefits of a multi-modal architecture and the resulting mode switches. The proposed *MDSL* provides a flexible and systematic way to define modes and to use different software products and versions from a security perspective. Moreover, it allows simulating events to test the mode switches' effectiveness. Within the Eclipse IDE, the user-defined system description is analyzed by a Xtext validator to ensure that names are unique and inheritance is properly used. Larger and more complex systems with multiple web servers, databases, and applications may require additional constraints and rules to be considered. Here we have planned support for a set of rules in the *MDSL*. In addition to the reaction of changed vulnerability scores, the framework could react to other events simultaneously, like an increasing or decreasing number of online users, or if there exist exploits for specific vulnerabilities. That can lead to the dynamic adaption of the system configuration or a mode switch.

As already mentioned in Section 3.4, software product line concepts can complement our modes. For example, we can build mode switches upon variability models and dynamic product configuration at runtime, and leverage capabilities to define features, dependencies between different components, or cross-tree constraints (Acher et al., 2011; Morin et al., 2008). Furthermore, providing a graphical user interface for defining the modes can complement the textual description of our *MDSL*. That would increase usability and allow the definition of modes and actions for users less familiar with programming language concepts.

7.2. Multi-modal framework

Based on our evaluation, we have observed that modes can be used to describe more than one version of, for example, a web server or different configurations of a system. Besides switching modes in parts of a system when affected by a vulnerability, our mechanism can also be used to adapt components on a more fine-grained level. For example, to change single properties of configurations to force secure login or adapt access control (Morin et al., 2010). In terms of further easing the creation of a system description with our *MDSL*, we can detect software versions automatically by scanning the target system, and support multiple versions of software. We plan to automatically detect the currently installed software versions to create a software inventory in the future. Furthermore, we want to extend our work with existing configuration management systems like *Ansible*, *Puppet*, or others to support several operating systems and simplify provisioning and application deployment on multiple systems. The patches were collected specifically for the Debian distribution and must be adapted when another operating system or distribution is used. In the future, we want to integrate modes not just as logical encapsulation of existing parts, but even more deeply into the software, i.e., into the kernel.

We envision that our framework will be used not only in systems with high security requirements. Using modes can help organizations to quickly adjust security measures based on changing conditions or evolving threats. In addition, it will help to gain better security posture visibility and improve risk management.

7.3. Self-adaptive systems and moving target defense

Our multi-modal framework not only provides a one-way step-by-step degradation functionality to increase security. It also includes a self-adaptive defense based on the vulnerability scores. In addition, we use a self-healing functionality, as the system switches back to a mode with more functionality after installing an update or patch. This mechanism can also be helpful in case of failures or problems after updates, e.g., the web applications are not working as expected. The system can then switch to another mode to provide business continuity. As an extension, switching between functional equivalent modes periodically or randomly makes attacks more complicated and relates to the idea of Moving Target Defense.

7.4. Mitigating vulnerabilities by switching modes

The case study results show that mode switching between all modes slightly improved security compared to the mode *Nginx*. One might suggest just using the web server nginx instead of Apache, because it had fewer vulnerabilities in history, see Table 3. However, mode switching resulted in three fewer days of the system being at risk, a 93.7 lower CVSS score sum, provided

more functionality, and added additional flexibility to react to future vulnerabilities we might not know today. Even one day more risk may be enough for state-sponsored attackers to take over a web application. In our case, the *Apache* mode was suggested seven times in August 2019 and in January 2020 because of a higher CVSS sum of the mode *Nginx*. Because CMSs like WordPress and Joomla need PHP, switching only between the corresponding PHP modes may be more practical. However, it may be better to provide a static HTML page than a vulnerable dynamically generated page with PHP, or an HTTP error page like *404 Not Found* in some cases. For example, online banking applications can switch to a secure mode without taking any risk if confidentiality and integrity are more important than availability. Our multi-modal architecture optimizes the mode selection and provides the optimal mode with the lowest total severity score. Needless to say, unknown vulnerabilities and exploits will still threaten systems. In addition, mode switching can have an even more significant impact if vulnerability information is provided immediately after reporting to the manufacturer. The system does not need all details to switch from one mode to another. It is sufficient to know that one mode is riskier than another. As already mentioned, IDS or a log file analyzer can help to detect attacks and can be another trigger to switch to a mode with lower risk. Considering exploit databases also help to reduce the attack surface. Vulnerability scanners such as *OpenVAS* (OpenVAS, 2022) can be used to (automatically) build up a software inventory and to check whether they are affected by CVEs periodically. Usually, these tools only inform system administrators. However, we propose to trigger an automatic mode switch in these cases. Mode switching will not be the silver bullet to solve all security problems for all possible systems. However, it will be a way to mitigate the critical period of time between when a vulnerability becomes known and when an update or patch is provided and eventually installed for systems where components and configurations can be exchanged.

We have to consider security aspects in all phases of development to identify security flaws at an early stage (Tuma et al., 2020; Cabot and Zannone, 2008). Nevertheless, it is still necessary to speed up the update and patch process. One extension of our work can be to consider the number of days a vulnerability exists and is not fixed, similar to the temporal risk strategy by Rapid7 (2021). In addition, new work on machine-readable CVE advisories (OASIS, 2020) can be added to decide whether a system is affected and the vulnerability exploitable. The concept of mode switching and our architecture is not limited to reported vulnerabilities, but can serve as a framework for reacting to potentially suspicious data or immanent security issues. For example, it can be used with various different early warning systems (Azzam et al., 2021; Apel et al., 2009) and intrusion detection solutions (Khazraei et al., 2021; Koley et al., 2021; Thakur et al., 2021; Aliabadi et al., 2021). Once a potential intrusion has been detected, or a warning is issued, our mode switching framework can provide the basis for transitioning to a different mode, thus, for example, deactivating the endangered component or system, or activating more restrictive security policies as part of a configuration change. Even vulnerability forecasts (Leverett et al., 2020) can be a trigger for a mode switch. Having two web servers installed and running, the *Mode Control* needs additional disk space and CPU time, but it provides a method to reduce the risk level of being hacked. Another important point is to prevent misuse of the mode-switching logic so as not to end up like *SolarWinds* (CISA, 2022) and others, where the security solution became a problem. For example, the *REvil* ransomware misused Windows Safe Mode in order to encrypt files on the hard drive (Abrams, 2021). We mentioned the benefits and drawbacks of a multi-modal architecture from a security point of view in Riegler and Sametinger (2021b).

7.5. Advantages and limitations

On the one hand, mode switching will increase system complexity, make the design and development phase more extensive and more expensive, and may lead to unintended negative side effects. Mode switching may increase the software evolution overhead with regression testing and impact analysis. Additional maintenance may be necessary to control the modes. False-positive vulnerabilities and automatic mode switching can also lead to unintended system behavior changes. However, insiders can misuse the mode-switching mechanism and the central control of multiple devices. Eventually, a CVE ignore list will be needed in the future. This list can be maintained by security professionals and/or by the operators. In addition, humans-in-the-loop can help to prevent unintended behavior.

On the other hand, mode switching will secure the time between vulnerability disclosure and security patch installation, reduce the attack surface, and make the system more resilient. With mode-switching, manual risk reduction is possible even if a system's or module's manufacturer has stopped service. For example, Cisco has not yet released and will not release software updates (Cisco, 2020) to address recently discovered vulnerabilities for end-of-life firewalls and VPN routers. We can ease the use of our framework by providing several sample modes for common systems, like our case study for web servers. Furthermore, modes can be used to reduce system downtime by switching to another mode during the update or maintenance of software. To reduce false-positive vulnerabilities and overreactions, we can manually inspect the vulnerabilities (human-in-the-loop) or use machine learning models. In addition, an internal vulnerability database can reduce dependency on public databases and allow a differentiated view. Finally, an authorization and monitoring system is essential to prevent and detect insider attacks, e.g., switching multiple devices to a degraded or more vulnerable mode.

8. Conclusion

In this paper, we have presented a model-driven mode-switching framework to protect systems resiliently. Administrators can reuse existing micro-scripts, describe modes with a *Mode Domain-Specific Language* (MDSL) on the macro level, and have the flexibility to switch among them. In addition, tool support and our framework provides syntax highlighting, code completion, validation for the MDSL, and generates code for different environments. We have shown its use with a web server scenario. Historic vulnerabilities of two years were used to compare risk scores and showed that mode switching reduces the window of exposure from 536 to 8 days. This has resulted in zero known risk (total CVSS score) in 98.9% of the analyzed time.

In the future, we want to combine our approach with existing configuration management systems such as *Ansible* or *Puppet*, and intrusion detection systems like *fail2ban* to further enhance scalability and usability. In the first step, we can use a simple log file analyzer or a file integrity checker to switch modes based on detected actual attacks of vulnerabilities currently unknown to the public. The MDSL will be extended with rules and events to provide more specific settings for a mode switch. We also aim to provide "template" mode definitions, such as the ones used in the case study for different operating systems, and investigate how practitioners can use the framework and integrate it into their software development process. In order to ward off even more targeted attacks, we plan to integrate modes deeper into the software and combine them with software product lines. Another interesting aspect is whether it is possible to predict vulnerabilities. Finally, we aim to monitor and control several multi-mode systems and investigate how mode switching will make industrial edge computing more secure and resilient.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

We provide our data, the MDSL, and the framework in a long-term archive:

Mode-Switching Framework (Zenodo).

Acknowledgments

This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria and by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development (CDG).

References

Abeni, L., Buttazzo, G., 2001. Hierarchical QoS management for time sensitive applications. In: Proceedings of the Seventh IEEE Real-Time Technology and Applications Symposium. pp. 63–72. http://dx.doi.org/10.1109/RTTAS.2001.929866.

Abie, H., Savola, R.M., Bigham, J., Dattani, I., Rotondi, D., Da Bormida, G., 2010. Self-healing and secure adaptive messaging middleware for business-critical systems. International Journal on Advances in Security 3 (1&2), http://www.iariajournals.org/security/sec_v3_n12_2010_paged.pdf.

Abrams, L., 2021. REvil Ransomware has a New 'Windows Safe Mode' Encryption Mode. Bleeping Computer, https://www.bleepingcomputer.com/news/security/revil-ransomware-has-a-new-windows-safe-mode-encryption-mode, [Last accessed: 2022-09-30].

Acher, M., Collet, P., Lahire, P., Moisan, S., Rigault, J.-P., 2011. Modeling variability from requirements to runtime. In: 16th IEEE International Conference on Engineering of Complex Computer Systems. pp. 77–86. http://dx.doi.org/10.1109/ICECCS.2011.15.

Aliabadi, M.R., Seltzer, M., Vahidi Asl, M., Ghavamizadeh, R., 2021. ARTINALI#: An efficient intrusion detection technique for resource-constrained cyber-physical systems. Int. J. Crit. Infrastruct. Prot. http://dx.doi.org/10.1016/j.ijcip.2021.100430.

Allier, S., Barais, O., Baudry, B., Bourcier, J., Daubert, E., Fleurey, F., Monperrus, M., Song, H., Tricoire, M., 2015. Multitier diversification in web-based software applications. IEEE Software 32 (1), 83–90. http://dx.doi.org/10.1109/MS.2014.150.

Apache, 2020. Important: Push diary crash on specifically crafted HTTP/2 header (CVE-2020-9490). The Apache Software Foundation, https://httpd.apache.org/security/vulnerabilities_24.html#CVE-2020-9490, [Last accessed: 2022-09-30].

Apel, M., Biskup, J., Flegel, U., Meier, M., 2009. Towards early warning systems–challenges, technologies and architecture. In: International Workshop on Critical Information Infrastructures Security. Springer, pp. 151–164. http://dx.doi.org/10.1007/978-3-642-14379-3_13.

Arcaini, P., Mirandola, R., Riccobene, E., Scandurra, P., 2018. A DSL for MAPE patterns representation in self-adapting systems. In: European Conference on Software Architecture. Springer, pp. 3–19. http://dx.doi.org/10.1007/978-3-030-00761-4_1.

Arcaini, P., Riccobene, E., Scandurra, P., 2015. Modeling and analyzing MAPE-K feedback loops for self-adaptation. In: Proceedings of 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. IEEE, pp. 13–23. http://dx.doi.org/10.1109/SEAMS.2015.10.

Autosar, 2017. Guide to mode management. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_EXP_ModeManagementGuide.pdf, [Last accessed: 2022-09-30].

Avgeriou, P., 2006. Run-time reconfiguration of service-centric systems. In: EuroPLoP 2006 - 11th European Conference on Pattern Languages of Programs. pp. 79–94, https://www.cs.rug.nl/~paris/papers/EPLOP06.pdf.

Azzam, M., Pasquale, L., Provan, G., Nuseibeh, B., 2021. Grounds for suspicion: Physics-based early warnings for stealthy attacks on industrial control systems. arXiv:2106.07980.

Barna, C., Khazaei, H., Fokaefs, M., Litoiu, M., 2017. Delivering elastic containerized cloud applications to enable DevOps. In: 2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS, IEEE, pp. 65–75. http://dx.doi.org/10.1109/SEAMS.2017.12.

Baudry, B., Monperrus, M., 2015. The multiple facets of software diversity: Recent developments in year 2000 and beyond. ACM Comput. Surv. 48 (1), http://dx.doi.org/10.1145/2807593.

Beek, M.H.t., Legay, A., Lafuente, A.L., Vandin, A., 2020. Variability meets security: qantitative security modeling and analysis of highly customizable attack scenarios. In: Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems. VAMOS '20, ACM, pp. 1–9. http://dx.doi.org/10.1145/3377024.3377041.

Brambilla, M., Cabot, J., Wimmer, M., 2017. Model-Driven Software Engineering in Practice, Second Edition. In: Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers, http://dx.doi.org/10.1007/978-3-031-02549-5.

Cabot, J., Zannone, N., 2008. Towards an integrated framework for model-driven security engineering. In: Whittle, J., Jürjens, J., Nuseibeh, B., Dobson, G. (Eds.), Proceedings of the 1st Modeling Security Workshop. In: CEUR Workshop Proceedings, CEUR-WS.org, http://ceur-ws.org/Vol-413/paper13.pdf.

Chen, T., Phan, L.T.X., 2018. SafeMC: A system for the design and evaluation of mode-change protocols. In: Proceedings of the 2018 IEEE Real-Time and Embedded Technology and Applications Symposium. pp. 105–116. http://dx.doi.org/10.1109/RTAS.2018.00021.

Cheng, B.H.C., Clark, R.J., Fleck, J.E., Langford, M.A., McKinley, P.K., 2020. ACROS: assurance case driven adaptation for the robot operating system. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. MODELS '20, ACM, pp. 102–113. http://dx.doi.org/10.1145/3365438.3410952.

Cheng, S.-W., Garlan, D., 2012. Stitch: A language for architecture-based self-adaptation. J. Syst. Softw. 85 (12), 2860–2875. http://dx.doi.org/10.1016/j.jss.2012.02.060.

Cimpanu, C., 2019. Google Project Zero: 95.8% of All Bug Reports are Fixed Before Deadline Expires. ZDNet, https://www.zdnet.com/article/google-project-zero-95-8-of-all-bug-reports-are-fixed-before-deadline-expires/, [Last accessed: 2022-09-30].

CISA, 2022. Emergency directive 21-01 | CISA. US Cybersecurity and Infrastructure Security Agency (CISA), https://www.cisa.gov/emergency-directive-21-01, [Last accessed: 2022-09-30].

Cisco, 2020. Cisco small business RV110W, RV130, RV130W, and RV215W routers management interface remote command execution vulnerability. https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-rv-rce-q3rxHnvm, [Last accessed: 2022-09-30].

Debian, 2020. Package tracker apache2 (2.4.38-3+deb10u4) buster-security; urgency=high. https://tracker.debian.org/media/packages/a/apache2/changelog-2.4.38-3deb10u4, [Last accessed: 2022-09-30].

Debian, 2022a. All information about the security bug tracker in JSON format. https://security-tracker.debian.org/tracker/data/json, [Last accessed: 2022-09-30].

Debian, 2022b. DebianReleases. https://wiki.debian.org/DebianReleases, [Last accessed: 2022-09-30].

Debian, 2022c. Security information. https://www.debian.org/security/index.en.html, [Last accessed: 2022-09-30].

Dsouza, G., Rodriguez, G., Al-Nashif, Y., Hariri, S., 2013. Building resilient cloud services using DDDAS and moving target defence. International Journal of Cloud Computing 2 (2/3), http://dx.doi.org/10.1504/IJCC.2013.055266.

Eclipse Foundation, 2020. Xtend - Modernized Java. https://www.eclipse.org/xtend, [Last accessed: 2022-09-30].

Eclipse Foundation, 2022. Xtext - Language engineering framework. https://www.eclipse.org/Xtext, [Last accessed: 2022-09-30].

Edgescan, 2019. Vulnerability stats report 2019. https://www.edgescan.com/wp-content/uploads/2019/02/edgescan-Vulnerability-Stats-Report-2019.pdf, [Last accessed: 2022-09-30].

Firesmith, D., 2019. System Resilience: What Exactly is it?. Carnegie Mellon University, Software Engineering Institute, https://insights.sei.cmu.edu/sei_blog/2019/11/system-resilience-what-exactly-is-it.html, [Last accessed: 2022-09-30].

FIRST, 2022. Common vulnerability scoring system SIG. Forum of Incident Response and Security Teams (FIRST), https://www.first.org/cvss, [Last accessed: 2022-09-30].

Gerostathopoulos, I., Skoda, D., Plasil, F., Bures, T., Knauss, A., 2019. Tuning self-adaptation in cyber-physical systems through architectural homeostasis. J. Syst. Softw. 148, 37–55. http://dx.doi.org/10.1016/j.jss.2018.10.051.

Glass, M., Lukasiewycz, M., Haubelt, C., Teich, J., 2009. Incorporating graceful degradation into embedded system design. In: Design, Automation Test in Europe Conference Exhibition. pp. 320–323. http://dx.doi.org/10.1109/DATE.2009.5090681.

Gomaa, H., Hussein, M., 2004. Software reconfiguration patterns for dynamic evolution of software architectures. In: Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture. pp. 79–88. http://dx.doi.org/10.5555/998676.999529.

Göttmann, H., Luthmann, L., Lochau, M., Schürr, A., 2020. Real-time-aware reconfiguration decisions for dynamic software product lines. In: Proceedings of the 24th ACM Conference on Systems and Software Product Line. ACM, pp. 1–11. http://dx.doi.org/10.1145/3382025.3414945.

Halstead, M., 1977. Elements of Software Science. In: Computer Science Library, Elsevier.

Hang, Y., Hansson, H., 2013. Handling multiple mode switch scenarios in component-based multi-mode systems. In: Proceedings of the 20th Asia-Pacific Software Engineering Conference. pp. 404–413. http://dx.doi.org/10.1109/APSEC.2013.61.

Hendrix, R., 2008. Aerospace system improvements enabled by modern phased array radar. In: IEEE Radar Conference. pp. 1–6. http://dx.doi.org/10.1109/RADAR.2008.4720770.

Hiesgen, R., Nawrocki, M., Schmidt, T.C., Wählisch, M., 2022. The race to the vulnerable: Measuring the log4j shell incident. arXiv preprint arXiv:2205.02544.

Hili, N., Bagherzadeh, M., Jahed, K., Dingel, J., 2020. A model-based architecture for interactive run-time monitoring. Softw. Syst. Model. 19 (4), 959–981. http://dx.doi.org/10.1007/s10270-020-00780-y.

Kephart, J., Chess, D., 2003. The vision of autonomic computing. Computer 36 (1), 41–50. http://dx.doi.org/10.1109/MC.2003.1160055.

Khakpour, N., Skandylas, C., Nariman, G.S., Weyns, D., 2019. Towards secure architecture-based adaptations. In: 2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS, IEEE, pp. 114–125. http://dx.doi.org/10.1109/SEAMS.2019.00023.

Khazraei, A., Hallyburton, S., Gao, Q., Wang, Y., Pajic, M., 2021. Learning-based vulnerability analysis of cyber-physical systems. arXiv:2103.06271 [cs, eess].

Knight, J.C., Strunk, E.A., 2004. Achieving critical system survivability through software architectures. In: de Lemos, R., Gacek, C., Romanovsky, A. (Eds.), Architecting Dependable Systems II. Springer, pp. 51–78. http://dx.doi.org/10.1007/978-3-540-25939-8_3.

Koley, I., Adhikary, S., Dey, S., 2021. An RL-based adaptive detection strategy to secure cyber-physical systems. arXiv:2103.02872 [cs].

Lee, J., Kang, K.C., 2006. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In: Proceedings of the 10th International Software Product Line Conference (SPLC'06). IEEE, pp. 10 pp.–140. http://dx.doi.org/10.1109/SPLINE.2006.1691585.

Lemos, R.d., Giese, H., Müller, H.A., Shaw, M., Andersson, J., Litoiu, M., Schmerl, B., Tamura, G., Villegas, N.M., Vogel, T., et al., 2013. Software engineering for self-adaptive systems: A second research roadmap. In: Software Engineering for Self-Adaptive Systems II. Springer, pp. 1–32. http://dx.doi.org/10.1007/978-3-642-35813-5_1.

Lerner, E., 2019. PHP-FPM + Nginx - Remote Code Execution. Exploit Database, https://www.exploit-db.com/exploits/47553, [Last accessed Sept. 9, 2021].

Leverett, E., Rhode, M., Wedgbury, A., 2020. Vulnerability forecasting: In theory and practice. arXiv:2012.03814 [cs].

Mirandola, R., Riccobene, E., Scandurra, P., 2019. Self-accounting in architecture-based self-adaptation. In: Proceedings of the 13th European Conference on Software Architecture-Volume 2. pp. 14–17. http://dx.doi.org/10.1145/3344948.3344957.

MITRE, 2022a. CPE - Common Platform Enumeration. The MITRE Corporation, https://cpe.mitre.org/, [Last accessed: 2022-09-30].

MITRE, 2022b. CVE - Common Vulnerabilities and Exposures. The MITRE Corporation, https://cve.mitre.org/, [Last accessed: 2022-09-30].

MITRE, 2022c. CWE - Common Weakness Enumeration. The MITRE Corporation, https://cwe.mitre.org, [Last accessed: 2022-09-30].

Moghadam, M.H., Saadatmand, M., Borg, M., Bohlin, M., Lisper, B., 2018. Adaptive runtime response time control in PLC-based real-time systems using reinforcement learning. In: 2018 IEEE/ACM 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). IEEE, pp. 217–223. http://dx.doi.org/10.1145/3194133.3194153.

Morin, B., Fleurey, F., Bencomo, N., Jézéquel, J.-M., Solberg, A., Dehlen, V., Blair, G., 2008. An aspect-oriented and model-driven approach for managing dynamic variability. In: Model Driven Engineering Languages and Systems. In: Lecture Notes in Computer Science, vol. 5301, Springer Berlin Heidelberg, pp. 782–796. http://dx.doi.org/10.1007/978-3-540-87875-9_54.

Morin, B., Mouelhi, T., Fleurey, F., Le Traon, Y., Barais, O., Jézéquel, J.-M., 2010. Security-driven model-based dynamic adaptation. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering - ASE'10. ACM Press, pp. 205–214. http://dx.doi.org/10.1145/1858996.1859040.

Mosser, S., Blay-Fornarino, M., Duchien, L., 2012. A commutative model composition operator to support software adaptation. In: Modelling Foundations and Applications. In: Lecture Notes in Computer Science, vol. 7349, Springer, pp. 4–19. http://dx.doi.org/10.1007/978-3-642-31491-9_3.

NASA, 2020. ULA launch mars 2020 perseverance rover mission to red planet. US National Aeronautics and Space Administration (NASA), https://mars.nasa.gov/news/8724/nasa-ula-launch-mars-2020-perseverance-rover-mission-to-red-planet, [Last accessed: 2022-09-30].

Nguyen, P.H., Ali, S., Yue, T., 2017. Model-based security engineering for cyber-physical systems: A systematic mapping study. Inf. Softw. Technol. 83, 116–135. http://dx.doi.org/10.1016/j.infsof.2016.11.004.

Nguyen, P.H., Kramer, M., Klein, J., Traon, Y.L., 2015. An extensive systematic review on the model-driven development of secure systems. Inf. Softw. Technol. 68, 62–81. http://dx.doi.org/10.1016/j.infsof.2015.08.006.

NIST, 2020. CVE-2020-9490 detail. US National Institute for Standards and Technology (NIST), https://nvd.nist.gov/vuln/detail/CVE-2020-9490, [Last accessed: 2022-09-30].

NIST, 2022a. National Vulnerablility Database (NVD). US National Institute for Standards and Technology (NIST), https://nvd.nist.gov/, [Last accessed: 2022-09-30].

NIST, 2022b. Retrieve CPE information. US National Institute for Standards and Technology (NIST), https://nvd.nist.gov/developers/products, [Last accessed: 2022-09-30].

NUREG, 1995. Standard Technical Specifications: Babcock and Wilcox Plants. Revision 1, no. NUREG–1430-Vol.1-Rev.1, 87064. US Department of Energy Office of Scientific and Technical Information (OSTI), Nuclear Regulatory Commission (NUREG), Washington, D.C., http://dx.doi.org/10.2172/87064.

OASIS, 2020. Common Security Advisory Framework (CSAF). https://oasis-open.github.io/csaf-documentation/, [Last accessed: 2022-09-30].

Offensive Security, 2022. Exploit database. https://www.exploit-db.com, [Last accessed: 2022-09-30].

OpenVAS, 2022. Open Vulnerability Assessment Scanner. https://openvas.org, [Last accessed: 2022-09-30].

Oracle, 2022. Critical patch updates, security alerts and bulletins. https://www.oracle.com/security-alerts/, [Last accessed: 2022-09-30].

Packet Storm, 2020. Apache 2.4.43 mod_http2 memory corruption. https://packetstormsecurity.com/files/160392/Apache-2.4.43-mod_http2-Memory-Corruption.html, [Last accessed: 2022-09-30].

Pedroza, G., Mockly, G., 2020. Method and framework for security risks analysis guided by safety criteria. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings. MODELS '20, ACM, pp. 1–8. http://dx.doi.org/10.1145/3417990.3420047.

Phan, L.T., Lee, I., 2011. Towards a compositional multi-modal framework for adaptive cyber-physical systems. In: Proceedings of the 2011 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications. pp. 67–73. http://dx.doi.org/10.1109/RTCSA.2011.82.

Poremba, S., 2020. Old Vulnerabilities Open the Door for WannaCry Ransomware. Security Boulevard, https://securityboulevard.com/2020/12/old-vulnerabilities-open-the-door-for-wannacry-ransomware/, [Last accessed: 2022-09-30].

Pradhan, S., Dubey, A., Levendovszky, T., Kumar, P.S., Emfinger, W.A., Balasubramanian, D., Otte, W., Karsai, G., 2016. Achieving resilience in distributed software systems via self-reconfiguration. J. Syst. Softw. 122, 344–363. http://dx.doi.org/10.1016/j.jss.2016.05.038.

Rao, A., Carreón, N., Lysecky, R., Rozenblit, J., 2018a. Probabilistic threat detection for risk management in cyber-physical medical systems. IEEE Softw. 35 (1), 38–43. http://dx.doi.org/10.1109/MS.2017.4541031.

Rao, A., Rozenblit, J., Lysecky, R., Sametinger, J., 2018b. Trustworthy multi-modal framework for life-critical system security. In: Proceedings of the Annual Simulation Symposium. Society for Modeling & Simulation International (SCS), pp. 17:1–17:9. http://dx.doi.org/10.5555/3213032.3213049.

Rapid7, 2021. Risk strategies - InsightVM documentation. https://docs.rapid7.com/insightvm/working-with-risk-strategies-to-analyze-threats/, [Last accessed: 2022-09-30].

Real, J., Crespo, A., 2004. Mode change protocols for real-time systems: A survey and a new proposal. Real-Time Syst. 26 (2), 161–197. http://dx.doi.org/10.1023/B:TIME.0000016129.97430.c6.

Riegler, M., Sametinger, J., 2020. Mode switching from a security perspective: First findings of a systematic literature review. In: Database and Expert Systems Applications - DEXA 2020 Workshops. Springer, pp. 63–73. http://dx.doi.org/10.1007/978-3-030-59028-4_6.

Riegler, M., Sametinger, J., 2021a. Mode switching for secure web applications – A juice shop case scenario. In: Database and Expert Systems Applications - DEXA 2021 Workshops. Springer, pp. 3–8. http://dx.doi.org/10.1007/978-3-030-87101-7.

Riegler, M., Sametinger, J., 2021b. Multi-mode systems for resilient security in industry 4.0. Procedia Comput. Sci. 180, 301–307. http://dx.doi.org/10.1016/j.procs.2021.01.167, Proceedings of the 2nd International Conference on Industry 4.0 and Smart Manufacturing (ISM 2020).

Riegler, M., Sametinger, J., Vierhauser, M., Wimmer, 2023. Supplemental material for a mode-switching framework with web server case study. Zenodo, https://doi.org/10.5281/zenodo.7603904, [Last accessed: 2023-02-06].

Ross, R., 2014. Assessing Security and Privacy Controls in Federal Information Systems and Organizations: Building Effective Assessment Plans. Tech. Rep. NIST Special Publication (SP) 800-53A, Rev. 4, National Institute of Standards and Technology, Gaithersburg, MD, http://dx.doi.org/10.6028/NIST.SP.800-53Ar4.

Ross, R., McEvilley, M., Oren, J., 2016. Systems Security Engineering: Considerations for a Multidisciplinary Approach in the Engineering of Trustworthy Secure Systems. Tech. Rep. NIST Special Publication (SP) 800-160, National Institute of Standards and Technology, Gaithersburg, MD, http://dx.doi.org/10.6028/NIST.SP.800-160.

Runeson, P., Höst, M., 2008. Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. 14 (2), 131. http://dx.doi.org/10.1007/s10664-008-9102-8.

Salgueiro, P.D., Abreu, S.P., 2010. A DSL for intrusion detection based on constraint programming. In: Proceedings of the 3rd International Conference on Security of Information and Networks. ACM, pp. 224–232. http://dx.doi.org/10.1145/1854099.1854145.

Sametinger, J., Rozenblit, J., Lysecky, R., Ott, P., 2015. Security challenges for medical devices. Communications of the ACM 58 (4), 74–82. http://dx.doi.org/10.1145/2667218.

Schneier, B., 2000. Full disclosure and the window of exposure. https://www.schneier.com/crypto-gram/archives/2000/0915.html#1, [Last accessed: 2022-09-30].

Shih, C.-S., Yang, C.-M., Su, W.-L., Tsung, P.-K., 2018. OSAMIC: Online schedulability analysis of real-time mode change on heterogeneous multi-core platforms. In: Proceedings of the 2018 Conference on Research in Adaptive and Convergent Systems. RACS '18, ACM, pp. 205–212. http://dx.doi.org/10.1145/3264746.3264755.

Thakur, S., Chakraborty, A., De, R., Kumar, N., Sarkar, R., 2021. Intrusion detection in cyber-physical systems using a generic and domain specific deep autoencoder model. Comput. Electr. Eng. 91, 107044. http://dx.doi.org/10.1016/j.compeleceng.2021.107044.

Thompson, M., Mendolla, M., Muggler, M., Ike, M., 2016. Dynamic application rotation environment for moving target defense. In: 2016 Resilience Week (RWS). pp. 17–26. http://dx.doi.org/10.1109/RWEEK.2016.7573301.

Thomson, M., Loreoto, S., Greg, W., 2010. Hypertext transfer protocol (http) timeouts. Internet Engineering Task Force (IETF), https://tools.ietf.org/id/draft-thomson-hybi-http-timeout-00.html, [Last accessed: 2022-09-30].

Tomić, I., Chen, P.-Y., Breza, M.J., McCann, J.A., 2018. Antilizer: run time self-healing security for wireless sensor networks. In: Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services. pp. 107–116. http://dx.doi.org/10.1145/3286978.3287029.

Tuma, K., Sion, L., Scandariato, R., Yskout, K., 2020. Automating the early detection of security design flaws. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. pp. 332–342. http://dx.doi.org/10.1145/3365438.3410954.

Van Landuyt, D., Pasquale, L., Sion, L., Joosen, W., 2021. Threat models at run time: the case for reflective and adaptive threat management. In: SEAMS'21: Proceedings of the 16th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. http://dx.doi.org/10.1109/SEAMS51251.2021.00034.

Vogel-Heuser, B., Fay, A., Schaefer, I., Tichy, M., 2015. Evolution of software in automated production systems: Challenges and research directions. J. Syst. Softw. 110, 54–84. http://dx.doi.org/10.1016/j.jss.2015.08.026.

Wang, B., Li, X., de Aguiar, L.P., Menasche, D.S., Shafiq, Z., 2017. Characterizing and modeling patching practices of industrial control systems. Proceedings of the ACM on Measurement and Analysis of Computing Systems 1 (1), 18:1–18:23. http://dx.doi.org/10.1145/3084455.

Wilhelm, F., 2020. Issue 2030: apache2: memory corruption in http2 push diary implementation. https://bugs.chromium.org/p/project-zero/issues/detail?id=2030, [Last accessed: 2022-09-30].

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2012. Experimentation in Software Engineering. Springer Berlin, Heidelberg, http://dx.doi.org/10.1007/978-3-642-29044-2.

Yin, H., Hansson, H., 2018. Fighting CPS complexity by component-based software development of multi-mode systems. Designs 2 (4), 39. http://dx.doi.org/10.3390/designs2040039.

Michael Riegler is a Ph.D. Student in the LIT Secure and Correct Systems Lab, Researcher at the Institute of Business Informatics – Software Engineering at the Johannes Kepler University Linz, Austria and a former Research Associate at the Department of Electrical and Computer Engineering at the University of Arizona, USA. He works on medical and industrial security and design and worked for several years for an industrial company. He received his B.Sc. and M.Sc. in Business Informatics focusing on Security Engineering and Management from the same university in 2011 and 2014. His email address is michael.riegler@jku.at.

Johannes Sametinger is an Associate Professor at the Institute of Business Informatics – Software Engineering and the LIT Secure and Correct Systems Lab at the Johannes Kepler University Linz, Austria. He holds a Dr. techn. in Computer Science from the same university. His research interests include software engineering and IT security with an emphasis on software security and medical device security. He has made several research visits to universities in the United States and Canada, including the University of Arizona. His email address is johannes.sametinger@jku.at.

Michael Vierhauser is a Senior Researcher at the LIT Secure and Correct Systems Lab at the Johannes Kepler University Linz, Austria. He holds a Master's degree in Software Engineering and Ph.D. in Computer Science from the Johannes Kepler University Linz. His current research interests include cyber-physical systems, safety assurance and and runtime monitoring. His email address is michael.vierhauser@jku.at.

Manuel Wimmer is Full Professor and Head of the Institute of Business Informatics – Software Engineering at Johannes Kepler University Linz, Austria. He received his Ph.D. and his Habilitation from TU Wien. Currently, he is also leading the Christian Doppler Laboratory on Model-Integrated Smart Production (CDL-MINT). In this context, he is developing modeling approaches for smart production facilities, as well as techniques for the continuous evolution of such systems based on production information gathered and analyzed at runtime through digital twins. His email address is manuel.wimmer@jku.at.