The Journal of Systems & Software 200 (2023) 111652

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Runtime software patching: Taxonomy, survey and future directions☆

Chadni Islam *[,1], Victor Prokhorenko[1], M. Ali Babar

University of Adelaide, Adelaide, SA 5005, Australia

ARTICLE INFO

Article history:
Received 9 March 2022
Received in revised form 19 December 2022
Accepted 20 February 2023
Available online 24 February 2023

Keywords:
Runtime patching
Patch granularity
Patch strategy
Live patching
Hot patching
Dynamic patching

ABSTRACT

Runtime software patching aims to minimize or eliminate service downtime, user interruptions and potential data losses while deploying a patch. Due to modern software systems' high variance and heterogeneity, no universal solutions are available or proposed to deploy and execute patches at runtime. Existing runtime software patching solutions focus on specific cases, scenarios, programming languages and operating systems. This paper aims to identify, investigate and synthesize state-of-the-art runtime software patching approaches and gives an overview of currently unsolved challenges. It further provides insights into multiple aspects of runtime patching approaches such as patch scales, general strategies and responsibilities. This study identifies seven levels of granularity, two key strategies providing a conceptual model of three responsible entities and four capabilities of runtime patching solutions. Through the analysis of the existing literature, this research also reveals open issues hindering more comprehensive adoption of runtime patching in practice. Finally, it proposes several crucial future directions that require further attention from both researchers and practitioners.

1. Introduction

Modern software is constantly evolving and adapting to ever-changing user needs and requirements, leading to the necessity to apply various changes to the existing software code. In the traditional software development life cycle, software changes are implemented in code compilation and deployment steps. These changes range from simple bug fixes to full-fledged software reworks. The increasing complexity of modern software and growing user demands lead to frequent software modifications that are provided in the form of updates and patches. Such code modifications typically attempt to improve or extend the existing software functionality to satisfy new user requirements. Improving does not necessarily relate to the core software system functionality; however, it can focus on enhancing the auxiliary properties such as security or privacy.

There is no universally accepted formal definition of the difference between patching and updating. However, a common understanding is that software changes that introduce new functionality are called *updates* (or upgrades). In contrast, minor changes that fix existing bugs or vulnerabilities are referred to as *patches*. Alternatively, some versioning systems[2] loosely define software compatibility as a dividing line between patches and updates. However, even minor bug-fixing patches may break the compatibility with the previous version rendering such compatibility-based distinction inadequate in practice. Regardless of the terminology used, both patches and updates essentially refer to **code modifications** which range from *simple* to quite *complex*.

In the simplest form, replacing the previously running software instance with an updated version involves stopping the old software instance and starting the new one (Hicks and Nettles, 2005; Chen et al., 2017; Zhou et al., 2020; Arnold and Kaashoek, 2009; Hayden et al., 2012). Such running software instances could be individual processes, web services, virtual machines or containers. However, depending on the extent of the change and type of the software, the update process may cause lengthy software service downtime, which can negatively impact end-users experience. From an end-user perspective, updates can also be perceived as either existing usage session interruptions or new session establishing delays. Such software service disruptions may cause severe negative consequences in highly-critical environments such as health or industrial control domains. In addition, highly loaded profit-oriented environments such as large data centers or stock exchanges may suffer direct financial losses from even short downtime.[3] A recent report by Gartner has indicated the cost of downtime could be $1–5 million at the

☆ Editor: Earl Barr.

* Corresponding author.
E-mail addresses: chadni.islam@adelaide.edu.au (C. Islam), victor.prokhorenko@adelaide.edu.au (V. Prokhorenko), ali.babar@adelaide.edu.au (M.A. Babar).

[1] The first and second authors contributed equally to this research.

[2] https://semver.org/

[3] https://www.upguard.com/blog/the-cost-of-downtime-at-the-worlds-biggest-online-retailer

https://doi.org/10.1016/j.jss.2023.111652
0164-1212/

higher end for just one hour (Gartner, 2019). Therefore, the notion of *runtime software patching* (in short runtime patching) is developed, which is also referred to as hot-patching, dynamic patching, or live patching (Zhou et al., 2020).

In an attempt to improve traditional (also known as offline) patching, runtime software patching aims to minimize or completely avoid any software operation interruptions. Runtime patching is commonly considered a technique for fast vulnerability mitigation (Hicks and Nettles, 2005; Arnold and Kaashoek, 2009; Chen et al., 2013, 2018). However, compared to simple traditional patching (i.e., source code patching), patching a running software poses significant challenges. Namely, preserving existing end-user activities (as well as compatibility with future ones) ranging from human input to complicated network-based communication sessions must be achieved for efficient runtime patching. In other words, the future behavior of the software must be affected by the applied patch, with currently executed activities and used data also needing to be adjusted accordingly to maintain compatibility. In some cases, runtime patching injects code into vulnerable programs to achieve a temporary fix (Chen et al., 2013, 2018). These temporary fixes are designed to immediately disable or replace vulnerable code to prevent the exploitation of the vulnerability. In the meantime, a developer can implement and test a proper update that actually fixes the bug, rather than just disabling the currently broken functionality. Note that additional non-technical supply chain requirements related to patch approval, testing, signing and distribution may introduce further significant patch adoption delays in practice (Chen et al., 2018).

Adopting runtime patching requires intimate knowledge of the software implementation details. For instance, converting existing internal functions or objects to be compatible with the updated software version requires a deep understanding of the object formats, structures and locations. However, no comprehensive runtime patching mechanisms have been implemented due to the technical diversity of underlying hardware, compilers, third-party libraries, and Operating Systems (OS). Existing runtime patching solutions target specific domains or scenarios such as IoT devices or virtual machines and hypervisors at datacenter scale (Chen et al., 2015; Salls et al., 2017; Mugarza et al., 2020; Zhang et al., 2019, 2014). These solutions typically focus on platform-specific challenges such as hardware limitations inherent to resource-constrained IoT devices. For instance, even storing the patched copy of software might not be possible under tight storage constraints.

Existing patching studies and reviews do not provide a formal approach to determining patch scale quantitatively, hence, leaving a gap in terminology related to patch and update differences. In addition, due to the high number and diversity of existing platform-specific patching solutions, a lack of comprehensive and formal understanding of general runtime patching strategies and required implementation capabilities such as approach applicability and expected potential disruptions are noticed. Moreover, the roles of different parties involved in runtime patching and their responsibilities in various patching process steps are also not explicitly considered.

Therefore, we review the state-of-the-art literature on software runtime patching from the deployment aspects to gain an understanding of existing runtime patching techniques and approaches. Based on the analysis, we propose a taxonomy (discussed in Section 3) that provides a comprehensive view of the runtime patching solutions from the perspective of patch granularity (Section 4), patch approaches (Section 5) and patch responsibilities (Section 6). We also identify the challenges and the research gaps in runtime patching and highlight the future research direction for further improvement in this field in Section 8.

The rest of this study is structured as follows. Section 2 discusses the software patching in general, followed by Section 3 diving further into an overview of runtime patching specifically. Various runtime patching granularities are examined in Section 4. A comprehensive view of existing runtime patch deployment strategies, workflow and capabilities is presented in Section 5. Section 6 contains a detailed discussion of parties involved in the patching process along with their corresponding roles and responsibilities. Further research opportunities addressing the open challenges identified are outlined in Section 8 with Section 9 concluding this study.

2. Background

A generic patch management process is shown in Fig. 1. Irrespective of the extent of a change, updating a software system can be performed offline or online. As can be seen from Fig. 1, patch preparation, delivering a patch, applying a patch, testing the patch and potentially removing the patch (if applicable) are the key steps. Techniques of patch preparation differ depending on whether the original source code is available. Source-code patches are essentially self-contained and logically united code edits to improve or fix certain functionality, bugs or vulnerabilities. Modern code version control systems typically track such edits as code commits. Having full access to application source code enables the development of comprehensive and flexible changes. However, the main downside of source-based patching is the requirement to recompile the whole application to include the required changes. Depending on the size of the software package, the recompilation step alone might take significant time.

In contrast, lack of source code access means that the binary executable must be altered directly. Therefore, binary-based patches essentially contain the bytes that need to be changed in the original binary executable. Compared to source code patching, binary patching generally takes a shorter time as no lengthy recompilation step is necessary. However, binary patching poses an additional challenge during the preparation steps. Namely, the addresses of the bytes to be altered must be located within the original binary.

While outside of the main scope of this review, it is worth noting that rollback planning is crucial for all changes occurring in a production system. It is typically assumed that rollback occurs when the new behavior does not match the expected changes. This assumption may mean that some downtime is inevitable. Thus, rollback procedures can be considered somewhat less constrained in their nature. When possible, however, maintaining operation is still a priority while the rollback is being conducted. Therefore, in general, a rollback operation is equivalent to a runtime patch in the opposite direction (new code back to old code). The same self-consistency considerations must be taken into account in both directions. The key difference is that a longer preparation is possible before a patch is applied compared to potential abrupt failures. Generally, a rollback procedure highly depends on the type of patch applied, application data involved, expected impact, timing constraints and amount of code. This review purports to focus only on the patch deployment steps, approaches and solutions.

2.1. Traditional software patching

Traditional or offline patching implies stopping the running software instance, potentially converting existing data and starting a new instance (Chen et al., 2015; Orso et al., 2002). As shown in Fig. 1, the first step in software patching is patch preparation. The prepared patch targets **source** or **binary levels** depending on whether the source code is available. Source-targeted patches

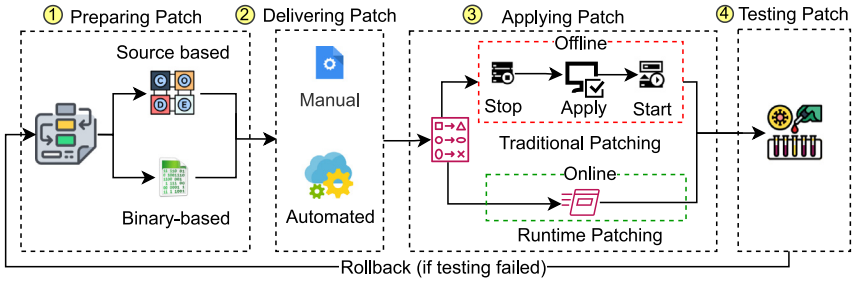


Fig. 1. The key steps of software patching life cycle.

require a potentially time-consuming software recompilation to produce a new (patched) executable, while binary-targeted modify the existing software executable. Source code level patching allows the programmer to easily modify arbitrary aspects of the software, such as replacing any functions, instructions and data flows. On the other hand, binary patches are somewhat more limited in their nature and are used for smaller-scale modifications (primarily in the security context). The main complication in binary patching is that any code length changes can cause significant memory pointers invalidation. In addition, in binary-level patching, finding the exact location to modify the buggy part may be complicated (Xu et al., 2020). Besides, the old binary can still run, further complicating the patching process. Furthermore, as the newly patched binary is ready for execution, running it may cause conflict due to resource sharing when the old binary uses the same resource.

Challenges with traditional patching: The major drawback of the traditional software patching approach is the associated **service interruption** when the old binary is stopped and the new one is not fully started (Hicks and Nettles, 2005; Orso et al., 2002). For instance, remote or local user sessions, network connections and data processing are interrupted or suspended when the patched software system stops. Short interruptions may be acceptable (albeit inconvenient) in interactive user sessions such as web browsing or word processing. Interactive software is commonly designed to automatically save and restore user sessions to mitigate such inconveniences to some extent. Web browsers that attempt to restore the state between restarts are a good example of such mitigation. In contrast, experiencing **service disruptions** in highly critical systems may cause significant monetary losses or be considered entirely unacceptable. For example, for life-support software and air-traffic controller, shutting down a system is prohibited or not considered an option (Orso et al., 2002). Thus, system administrators may opt to delay patch deployment to avoid potential system reboots that cause service disruption and loss of application state. For example, updating the OS on servers for highly interactive activities like online gaming or video streaming typically needs to schedule server downtime. During this time, players have to stop the game, wait for the servers to be updated and restarted, then login back in to the server and potentially start the game from the beginning, which is bothersome for the players. Unfortunately, keeping a vulnerable system un-patched for a prolonged period of time increases the risk of having the system exploited. In efforts to overcome these challenges of traditional patching, runtime patching has emerged and gained popularity. Details of runtime patching and related benefits are discussed in Section 3.

2.2. Existing reviews

Several software patch and update related reviews have been conducted in recent years (Seifzadeh et al., 2013; Ahmed et al., 2020; Miedes and Muñoz-Escoí, 2012; Lopez et al., 2017; Ilvonen et al., 2016; Gregersen et al., 2013; Mugarza et al., 2018). Ahmed et al. (2020) have performed a comprehensive systematic mapping study related to runtime software updating solutions. We encountered somewhat unstructured approach classifications while analyzing a large number of papers. For example, the most cited updating approaches include "Java VM" and "Multi-version", which do not necessarily have to be mutually exclusive. Furthermore, while the study has provided the statistics and an overview of approaches used, it lacks details on approach-specific benefits and challenges and the correlation between the adoption of techniques, tools and algorithms.

A more structured categorization of runtime software updating solutions is presented by Seifzadeh et al. (2013). This study presents a comprehensive, albeit high-level set of runtime updating evaluation metrics, such as scope, time of update, and type safety. Rather than focusing on existing implementations, Miedes and Muñoz-Escoí (2012) have outlined the concepts and techniques used in runtime software updating in general. In addition, the set of goals and requirements such as service continuity and generality are identified and discussed. However, no coherent taxonomy or classification of such approaches is presented, leading to somewhat mismatched categories. For instance, while being orthogonal in terms of goals, Java-oriented approaches and rollback-ability are discussed alongside technical challenges.

In addition, some reviews focus on particular technical domains and usage scenarios. For instance, Lopez et al. (2017) have surveyed existing function and system call hooking approaches. While function hooking can be used for a multitude of purposes (including malicious), function-level patching can significantly benefit by applying hooking techniques. One of the main strengths of the survey is the comprehensive view of both function and system call hooking under major operating systems. However, the scope of this review is limited to function granularity only and does not consider other patching levels. Another study by Gregersen et al. (2013) have compared three existing runtime patching implementations for Java applications. On top of evaluating the performance of the implementations, low-level patching capabilities were analyzed. The scope of the review is only limited to Java-specific capabilities such as class modifications were considered. Lastly, Mugarza et al. (2018) have focused on runtime software updating in the industrial IoT domain. Specifically, the requirements of safety systems are evaluated using nuclear control systems as a case study.

Unlike the existing reviews, Ilvonen et al. (2016) have taken an interesting perspective by analyzing the support of runtime or Dynamic Software Updating (DSU) in the software engineering education context. In particular, existing software engineering courses are analyzed to determine the adoption and coverage of DSU concepts. The main finding of this study is the lack of a holistic approach towards DSU in education, with only certain individual aspects being addressed in education.

Several observations are made based on the reviewed studies. Firstly, there is still a lack of common understanding in the domain of runtime patching, even at the level of terminology.

Secondly, no clear taxonomy coherently categorizing the existing runtime patching approaches has been developed so far. Thirdly, lack of generality hinders the wider adoption of the existing runtime software patching methodologies and tools. Fourthly, evaluation metrics vary wildly depending on the intended runtime patching domain, ranging from generic time overhead to language-specific class modifications.

Scope of our survey: Our survey focuses on multiple aspects of runtime patching approaches such as patch scale, general tactics and responsibility. A detailed taxonomy is presented along with the corresponding analysis of the existing solutions. The main focus lies within the *applying patch* phase in the patch management life cycle shown in Fig. 1. Unlike the existing surveys, which are typically narrow-focused and mainly highlight the methodologies and tools used for patching, we attempt to generalize the issues and approaches inherent to different patch granularity levels. It is worth noting that our survey does not compare the performance of different techniques or solutions because of the vastly different experimental setups and execution environments observed. Similarly, due to the high diversity of the solutions reviewed, different evaluation techniques cannot be compared directly, as different implementations focus on different metrics (e.g., downtime, overhead and long-term patch continuity). Naturally, systems aiming to achieve zero downtime would not even consider such a metric in their evaluation procedures. Furthermore, we indicate the existing state-of-the-art solutions' common challenges and suggest a set of future directions to advance the field.

3. Runtime patching

Runtime software patching aims to update a given software system while preserving running processes and sessions. In case some downtime is inevitable, runtime patching approaches focus on minimizing the disruption time. Zhou et al. (2020) has defined runtime patching as "*a method for dynamically updating software, effectively reducing the downtime and inconvenience often associated with software upgrades*". In contrast to traditional patching, runtime patching is primarily binary-oriented because the running binary instance of a program is modified in memory. The binary representation of the code needs to be replaced at runtime with the new (i.e., patched) version (Kashyap et al., 2016; Zhou et al., 2020). In addition to patching the in-memory version of a binary, a disk copy must be patched correspondingly so that patched behavior persists in any future restarts.

Runtime patching must take care of the current state that needs to be transformed to be compatible with the new code (Zhao et al., 2016; Chen et al., 2007; Pina and Hicks, 2016a) that includes in-RAM objects, data structures and external OS resources. A set of existing approaches address various aspects of running state transformation such as update points and state transformers (Chen et al., 2013, 2018; Giuffrida et al., 2013). In such approaches, the patch consists of a combination of the new code, safe update points and necessary state transformers. Update points are essentially the time windows suitable for applying a runtime patch. Data-specific state transformers can then transform the current program state to a new version. Specifically, a runtime patching system continuously monitors the program execution. When (if) a program reaches a suitable update point, the system loads the patched code and starts the program state transformation according to the specified state transformers. Once the transformation is complete, program execution continues with the new version being active (Zhao et al., 2016).

In severe cases, typically for large and complex systems, the state transformation may take considerable time, inducing noticeable service disruptions. For instance, transforming an existing Virtual Machine (VM) to make it compatible with a new hypervisor version might involve converting the virtual disk format. Given the multi-gigabyte sizes of modern disks, this operation might take a significant amount of time. Furthermore, not stopping the VM during the conversion could cause the disk contents to change before the new disk image is finalized. In such cases, runtime patching may choose to perform the conversion in multiple steps and only stop the VM during the last data portion conversion.

3.1. Goals and benefits of run-time patching

Runtime software patching approaches provide a number of practical benefits. First, from the security perspective, patching a running software reduces the vulnerable time window while a long-term solution is being prepared. A typical example of such immediate fixes is temporarily disabling a vulnerable code path, with the long-term solution to fix the expected code behavior. In some cases, applying such simple patches gain an additional time necessary for long-term testing changes at the expense of reduced functionality. Some prominent goals and benefits of runtime patching include reducing *service interruption* (Hicks and Nettles, 2005; Salls et al., 2017; Payer et al., 2013), *system downtime* (Zhou et al., 2020; Salls et al., 2017), *frequency of reboots* (Payer and Gross, 2013; Arnold and Kaashoek, 2009; Zhang et al., 2019) and *human involvement* (Salls et al., 2017; Arnold and Kaashoek, 2009). While achieving these goals, a runtime patching solution also aims to avoid loss of data and running states of applications to improve end-user experience (Zhou et al., 2020; Kashyap et al., 2016; Doddamani et al., 2019; Arnold and Kaashoek, 2009). In combination, these advantages of runtime software patching are beneficial from a business perspective in terms of service reliability and reduction of potential monetary losses. Reviewing the existing study, we identify that *high availability*, *quick vulnerability mitigation* and *improving user experiences* are three key aspects in consideration behind runtime patching.

- **High availability**: Providing uninterrupted highly-available service is crucial in critical domains such as defense, medical, industrial control systems and cloud systems. As such mission-critical systems require high availability (Hicks and Nettles, 2005; Chen et al., 2013, 2015; Zhao et al., 2016; Zhang et al., 2019), any disruptions caused by applying patch are unacceptable.
- **Quick vulnerability mitigation**: Runtime patching allows to mitigate vulnerabilities by fixing or disabling unsafe code fragments on the fly (Chen et al., 2013, 2017, 2018; Arnold and Kaashoek, 2009). In addition to full-fledged logic improvements, quick temporary solutions such as disabling vulnerable code paths or filtering unsafe user input can be used in practice.
- **Improving user experience:** Applying proper long-term runtime patches reduces the number of software system restarts, thus effectively simplifying overall system maintenance. In addition, this improves the end-user experience as fewer user workflow disruptions (including potential recovery steps) would be encountered during day-to-day system usage (Hicks and Nettles, 2005; Chen et al., 2017; Xu et al., 2020). From an end-user perspective, runtime patching attempts to reduce either frequency or length of service disruptions.

3.2. Proposed taxonomy

We propose a taxonomy to categorize the studies related to runtime patching as shown in Fig. 2 in terms of *what*, *how* and

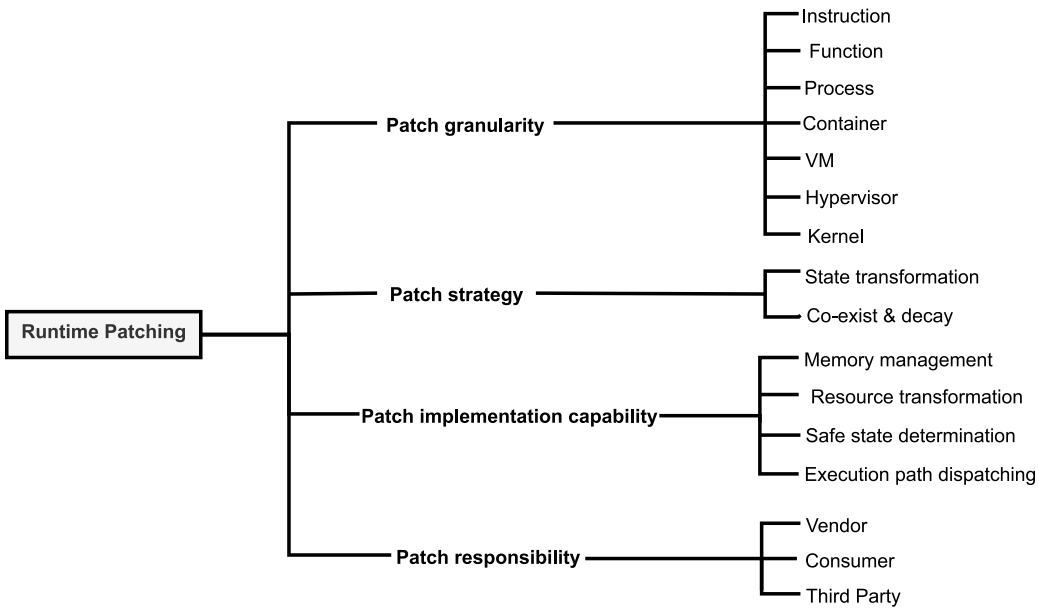


Fig. 2. Taxonomy of runtime patching.

who. While analyzing the patching techniques incorporated in different studies, we identify that patches are applied at different levels, such as individual function, single process or the whole Operating System (OS) kernel. We further find that a number of patching levels are implemented in the existing approaches. Hence, the first category we have in our taxonomy is **patch granularity**. By patch granularity, we mean the scope of a patch supported by a given solution. The scope of a patch refers to the scale of the change supported by a patching solution. For instance, some solutions support patching individual machine-level *instructions*, while others support replacing whole *functions*. On the other hand, larger-scale approaches focus on updating higher-level units such as whole *containers* or *VMs*.

We notice that when considering what to patch, the surveyed studies have addressed different issues and proposed different approaches. Therefore, analyzing the studies, we extract seven granularity levels by grouping the related runtime patching studies in terms of granularity that the studies aim to patch at runtime. Note that only a few papers are categorized into more than one sub-category for cases where one study has proposed an approach that performs patching at multiple granularities (Chen et al., 2015).

The second and third categories focus on the "how" aspect of the patching and the general techniques employed by an approach (**patch strategy**) and technical capabilities of a given patching system implementation discussed in Section 5. Two main strategies identified are *state transformation* and *co-exist & decay*. In some cases, direct state transformation is possible when the changes imposed by a patch are not significant. For instance, adding an object property or a method to reflect new (patched) functionality could be straightforward. In contrast, removing or modifying existing functions is likely to cause other code fragments that rely on previous behavior. Thus, the co-exist & decay approach aims to separate old and new data objects based on sessions or transactions where possible. In other words, old (unpatched) objects currently in use are not modified, while new code patches would be directed at updated objects. Later, old objects are disposed of when the pre-existing sessions/transactions are completed. The **implementation capabilities** directly reflect the practical applicability of a given solution. For instance, some systems might be capable of automated safe updating time windows detection, while others lacking such capability have to resort to manual assistance from the developers. Similarly, less capable systems may require extra external assistance in memory management to load or fit patched code into RAM.

In the fourth category, we have identified the entities involved and their **responsibilities** considered by a patching system. Specifically, we focus on "who" is responsible for applying the patches. The existing studies take three common responsibility-targeted views. First, *vendor-supported* patching systems that imply the original software system developers as in charge of the patching process. Second, *software system end-users* who need to patch a running system. Lastly, independent *third-party patchers* who attempt to provide facilities to apply generic patches to existing generic software (within certain limits). These situations differ in the amount of prior knowledge available to the patchers. Most significantly, developers would naturally access the original software source code, while end-users would not. Similarly, original developers would possess more profound knowledge of the application internals and logic.

There can be other ways the studies can be categorized; however, our primary focus is to identify the level at which a patch is applied and the adopted or proposed approaches to apply the patch at runtime. This helps us determine the issues of runtime patching approaches being neglected by the practitioners and identify potentially beneficial future directions.

4. Runtime patching granularities

This section covers the intended *patch granularity* in runtime patching that essentially characterizes the scale and boundaries of a patch to be applied. Patch granularity defines a responsibility boundary within which no state transformations are typically required or performed. The studies we have identified are categorized into seven groups: Instruction-level, Function-level, Process-level, Container-level, VM-level, Hypervisor-level and Kernel-level. Fig. 3 provides an overview of the objects or elements involved in patching at different granularities. While most straightforward patches target changing individual machine instructions in RAM, complex patches target replacing a vulnerable VM with a fixed one. For instance, Fig. 3 shows that patching an instruction requires taking memory pointers while patching a Host OS requires modifying the physical hardware state into account. Categorizing studies in terms of patch granularity gives

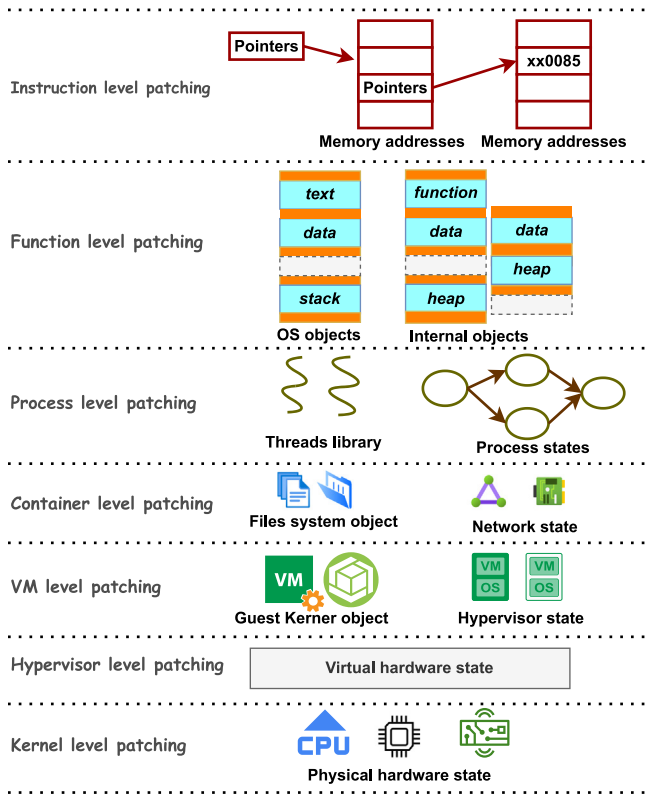


Fig. 3. Granularity-specific runtime assets, objects and patch boundaries. Patching at a lower layer of granularity requires transforming or dispatching the assets and objects on the same and all above layers.

insight into the objects and elements that require modifications when applying patches. It further helps understand the impact of a patch (i.e., the extent of change/patch consequences). For instance, changing an object in a lower layer, e.g., in a physical hardware state, impacts the higher layer object, such as file system and OS objects.

The patch preparation and deployment process highly depends on the patch granularity. For instance, replacing a few CPU instructions involves merely compiling the corresponding code into a target CPU architecture, whereas updating a VM may require rebuilding the whole VM image. Technical issues related to each granularity are discussed below. Unless significantly affecting the overall function behavior, patching individual instructions only takes care of the instruction abstraction level. Similarly, replacing a function with a patched variant needs to focus on the function abstraction level unless the overall process behavior change is externally observable. In contrast, switching over to a new kernel would need to take care of all processes that are using old kernel resources. Essentially, choosing a suitable patch granularity is a trade-off between transforming internal and external states. As shown in Fig. 3, connectivity points outside of the set boundary such as network sockets, external function calls and hardware may need to be adjusted to match the patched code.

4.1. Instruction-level patching

Instruction-level patching requires the patching solution to write the changed instructions into an appropriate memory location. It aims to replace machine-level CPU commands or individual bytes in the RAM (Chen et al., 2017; Xu et al., 2020; Chamith et al., 2016). We have identified a set of studies that have performed instruction level patching (Chen et al., 2017; Xu et al., 2020; Chamith et al., 2016; Chen et al., 2018, 2013, 2015). Patch preparation slightly differs in instruction level patching depending on the source code availability for a given software. In both cases, the patch preparation output is a sequence of bytes to be written to a certain memory address (potentially computed dynamically). Instruction level patching is lightweight and faster than function level patching (Chen et al., 2015).

Table 1 summarizes the works that have proposed instruction-oriented runtime patching approaches. We have identified the goal of each study along with the task related to patchings, such as patch generation, deployment, verification and restoration. By goal, we consider the focus on the studies — for instance, whether a solution is focused on updating software, fixing a bug, vulnerabilities or patching specific attacks such as buffer overflow. Table 1 further shows whether a patching task is fully automated, needs manual input from humans to trigger an automated task (i.e., semi-automated) or fully depends on humans. In most of the instruction level patching solutions, patch deployment is followed by patch generation, which are done automatically (Chen et al., 2013; Xu et al., 2020; Chen et al., 2017; Chamith et al., 2016). We have further identified the scope of the patching. Due to variations in technology and languages, a patching solution is not universal; thus, the scope of a particular study shows if the proposed approach is for Linux, a particular language like C or a cloud system. From Table 1, we observe that most of the instruction-level patching approaches are automated and span among several systems and application domains.

Instruction-level patching is performed by either inserting new instructions or modifying existing instructions. Chen et al. (2015) have considered both scenarios. In one scenario extra jump instructions are added. In another scenario, patches modify existing instructions to enable boundary condition safety checks. As shown in Table 1, a set of studies has used instruction level patching to mitigate software vulnerabilities (Chen et al., 2013; Payer et al., 2013; Chen et al., 2015). For instance, Chen et al. (2013) have proposed to monitor program instructions to check stack buffer overflow attacks. Here, instead of replacing a function, the designed patch application targets the specific instructions that trigger buffer overflows. They have proposed to automate the patch process from diagnosis to applying patches at runtime without interrupting the execution of the ongoing service. The proposed approach tracks function calls, however, it does not replace the whole functions and also does not consider cross-function changes. In another study, Chen et al. proposed to perform instruction-level patching for security patches that do not modify function argument types or amount (Chen et al., 2015). Arnold and Kaashoek (2009) have also proposed to patch individual instructions, however, they mainly replace entire functions. Payer et al. (2013), Payer and Gross (2013) have proposed a novel approach to combine sandbox with a runtime patching approach to maintain the integrity and availability of a system.

Similar to the above works, instruction level patching is also applied in Android OS (Chen et al., 2017, 2018; Xu et al., 2020) and Linux kernel (Rommel et al., 2019a). In Instaguard, Chen et al. (2018) have considered the function name to easily locate the memory addresses that trigger vulnerability exploitation, however, they do not replace the whole function with a patched version. They considered fine-grained line-oriented changes. In KARMA (Chen et al., 2017), the main target is to protect the integrity of the Android kernel whereas Instaguard protects the kernel from exploits originating from user space. Malicious inputs are filtered to prevent vulnerable kernel code from being exploited (Chen et al., 2018). In another study, Xu et al. (2020) have emphasized on automatic generation of patches and proposed an approach, namely Vulmet, that automatically generates

Table 1
Instruction-level patching.

Study	Goal	Task perform	Task type	Scope
D-linking (Hicks and Nettles, 2005)	Updating software	Patch generation and deployment	Semi-automated	C-like language
Safestack (Chen et al., 2013)	Buffer overflow attacks	Patch diagnosis, generation and deployment	Automated	Cloud system and data center
Instaguard (Chen et al., 2018)	Vulnerabilities in Android Program - integer overflow, buffer overflow, out-of-hour accesses and logic bugs	Patch generation and deployment	Semi-automated	Android System (Binaries) (Nexus 5)
Replus (Chen et al., 2015)	Security patches and long lived function	Patch generation and deployment	Semi-automated	Program in C language
Vulmet (Xu et al., 2020)	Vulnerabilities in Android kernel	Patch generation and deployment	Automated	Mobile system
Dynsec (Payer and Gross, 2013; Payer et al., 2013)	Vulnerability in Application (e.g., Apache web server)	Patch generation and deployment	Semi-Automated	Networked system
KARMA (Chen et al., 2017)	Vulnerabilities in Android kernel	Patch integrity verification and deployment	Automated	Android Kernel
WordPatch (Chamith et al., 2016)	x86 architecture probe toggling	Patch deployment	Automated	Processor
Kgraft (Suse, 2021)	Vulnerabilities	Patch deployment	Automated	Linux Kernel

patches for Android kernel vulnerabilities. Vulmet automatically computes necessary memory location in a function and replaces the vulnerable code with the specified patch. Instruction-level patching techniques are also used for investigating the effects of thread safety issues and analyzing the reaction and visibility of other execution threads on modifying x86 code (Chamith et al., 2016).

4.2. Function-level patching

Function-level patching is coarser compared to instruction-level patching as it targets patching whole functions (Arnold and Kaashoek, 2009; Lee et al., 2020). For example, it replaces a buggy function with a patched one to eliminate vulnerabilities. A set of studies have proposed to perform function-level patching (Arnold and Kaashoek, 2009; Lee et al., 2020; Duan et al., 2019; Jeong et al., 2017; Chen et al., 2015, 2007; Araujo and Taylor, 2020; Zhao et al., 2016; Rommel et al., 2019a). Performing function-level patching requires taking cross-function dependencies into consideration. In addition, long-living functions pose further challenges, as a suitable patching time must be determined. The steps of locating the patch target address are not much different compared to instruction-level patching. Table 2 summarizes the studies related to function-level patching. It shows most of the function-level patching studies, patch generation is performed before patch deployment. Similar to instruction-level patching the scope of the studies span different domain, OS and application.

Several of the runtime patchings approaches at function-level are designed for security patches (Arnold and Kaashoek, 2009; Lee et al., 2020; Duan et al., 2019; Hu et al., 2019). For instance, Duan et al. (2019) have proposed OSSPatcher that automatically identifies vulnerable functions, generates binary patches and performs patch injection at runtime. Similarly, Lee et al. (2020) have proposed an Appwrapper toolkit to inject additional security code on a per-method (i.e., function) basis into insecure apps enabling the use of dynamic policies to enhance overall application security. Function-level patching is also performed to replace a whole vulnerable function by linking it with a new function or replacement code into the kernel. The proposed method is considered very useful for large server environments that are highly utilized by multiple users (Arnold and Kaashoek, 2009). Research is also seen in patching vulnerabilities of binary programs via code transfer and binary rewriting in functions (Hu et al., 2019). Table 2 shows most of the function level patching is performed automatically without user intervention.

Existing works on patching kernel at runtime rely on the host Kernel and consider the Kernel to be always trusted. However, Zhou et al. (2020) have emphasized the fact that Kernel can be malicious and untrusted. Thus, they have proposed a reliable kernel runtime patching framework (albeit at function granularity) leveraging Trusted Execution Environments (TEE) that is hardware-assisted to prepare and deploy kernel patches. The proposed approach Kshot, prepares and deploys patches that do not need a trustworthy kernel patching mechanism (Zhou et al., 2020).

Considering the limited capacity of the embedded devices (e.g., lack of update functionality), Salls et al. (2017) have proposed an approach, Piston, to perform remote patching on embedded devices. Piston mainly performs patching at the function level. It is designed to force patches onto a vulnerable system by exploiting the vulnerabilities and taking control of the vulnerable process. For some vulnerabilities like stack-based buffer overflows, it supports automated patching whereas for others it is semi-automated and requires input from the analyst. Similarly, Ruckebusch et al. (2016a) have proposed a tool Gitar to enable runtime patching of applications in OSes running on IoT, M2M and resource-constrained devices. Table 2 shows that function-level runtime patching approaches are also adopted to perform dynamic variability in software systems (Rommel et al., 2019a; Rothberg et al., 2016). Most of these approaches are compiler assisted and implement a function multiverse in the compiler to perform binary patching.

Function-level patching is noticed to perform runtime patching at the production level, especially for C program (Chen et al., 2015). The authors have proposed a framework, Replus, to build environment-aware patches by separating the responsibilities of developers and customers where patch generation is performed in the developer environment and deploying or applying the patch is performed in a customer environment. The main focus of Replus is to provide a practical and efficient runtime patching system that does not need compiler support. Different from Replus, Orso et al. (2002) have proposed a tool, DUSC, to perform runtime patching in Java-based applications.

Table 2
Function-level patching.

Study	Goal	Task perform	Task type	Scope
Kshot (Zhou et al., 2020)	Patching vulnerabilities	Patch generation and deployment	Semi-automated	Linux Kernel
Replus (Chen et al., 2015)	Updating software	Patch generation and deployment	Automated	Program in C language
DUSC (Orso et al., 2002)	Updating Java based Software	Patch generation and deployment	Automated, Semi-Automated	Java based application
CURE (Zhao et al., 2016)	Generating safe patches and updating software	Patch generation and deployment	Automated	–
POLUS (Chen et al., 2007)	Updating software	Patch generation and deployment	Semi-automated	Contemporary server software
Piston (Salls et al., 2017)	Vulnerabilities in the embedded device - stack-based buffer overflow, heap overflow	Patch generation and deployment	Automated, Semi-automated	Embedded device - remote patching
Ksplice (Arnold and Kaashoek, 2009)	Patching vulnerabilities	Patch deployment	Automated	Linux Kernel
Multiverse (Rommel et al., 2019a)	Dynamic variability in performance critical paths	Patch deployment	Semi-automated	Cloud system, legacy code base
AppWrapper (Lee et al., 2020)	Applying dynamic security policies security functions	Patch deployment	Automated	Mobile devices (Android)
OSSPatcher (Duan et al., 2019)	Patching Vulnerabilities	Patch generation and deployment	Automated	Vulnerable mobile applications
Hotpaccher (Jeong et al., 2017)	Updating software	Patch deployment	Automated	ELF binary application
C-MultiVerse (Rothberg et al., 2016)	Dynamic variability in system software	Patch generation and deployment	Automated	Cloud system, legacy code base
Gitar (Ruckebusch et al., 2016a)	Updating network stack of constrained devices	Patch deployment	Automated	IoT, M2M, constrained devices' single Rime modules
BinPatch (Hu et al., 2019)	Patching Vulnerabilities in binary program (file level)	Patch generation, deployment and restoration	Automated	Binary code
UpdateCalculus (Bierman et al., 2003)	Formally Verifying patches before deployment	Patch verification	Manual	Theoretical

4.3. Process-level patching

Process-level patching aims to replace the whole process with a new one that contains a fixed or updated functionality (Mugarza et al., 2020; Giuffrida et al., 2013; Hayden et al., 2012; Ramaswamy et al., 2010). In contrast to function-level patching, updating the whole process might be easier in some cases as only OS dependencies and resources need to be tracked. For instance, files and network sockets need to be detached from the old process and transferred to the new process (Rommel et al., 2019b). The main complication in this area is maintaining the internal resource states such as current file pointers and network protocol states. Table 3 shows the key studies that have considered runtime patching at the process level. Among the existing studies, Rommel et al. (2019b) have proposed to move individual threads to new address space through predefined quiescent states. The proposed approach attempts to minimize waiting time by preparing the clone of the address space as opposed to the stop-the-world approach.

As shown in Table 3, we observe that most of the process-level patching solutions are semi-automated; hence, actual patch deployment requires human assistance. Process-level patching studies focus mainly on patch deployment rather than patch generation. The scope of the process-level patching solution typically targets a specific OS or programming language. A common way to achieve process-level patching is the usage of multi-version execution techniques (Hayden et al., 2012, 2014). Both update-safe points and corresponding data transformation functions must be predefined by the developers to make runtime updating possible. One of the proposed approaches, Kitsune (Hayden et al., 2012, 2014), implements single- and multi-threaded C-based application updating. The authors propose a novel tool to assist software developers in generating state transformation and transfer routines. Some light annotation is, however, still required from the developers. MVEDSUA (Pina et al., 2019) is an extension of Kitsune (Hayden et al., 2012) with the use of N-version execution framework Varan,[4] (Hosek and Cadar, 2015). The key insight behind MVEDSUA is the ability to patch a second copy

[4] Varan (Hosek and Cadar, 2015) is a rich framework in terms of applicability and is particularly useful for software updating purposes. The initial motivation behind the proposed approach is to run multiple copies of the same code with various modifications to minimize potential vulnerability risks. For instance, transparent failover can be achieved if only one of the copies is exploited. In the context of software updating, the same principle can be used to detect behavior deviations caused by the updated code as, for example, done in Pina et al. (2019).

Table 3
Process-level patching.

Study	Goal	Task perform	Task type	Scope
Cetratus (Mugarza et al., 2020)	Updating software securely	Patch deployment	Semi-automated	Safety-critical systems
Proteos (Giuffrida et al., 2013)	Updating software safely and stably	Patch deployment	Semi-automated	Custom OS (Minix based)
WaitFree (Rommel et al., 2019b)	High performance, Multi-threading support, Read-only memory segments	Patch deployment	Semi-automated	Linux on AMD64
MVEDSUA (Pina et al., 2019)	Updating software	Patch deployment and testing and rollback	Semi-automated	C-based application
TEDSUTO (Pina and Hicks, 2016b)	Updating software safely	Patch deployment and verification	Semi-automated	Java-based system
Katana (Ramaswamy et al., 2010)	Updating critical security or functionality	Patch generation, deployment	Automated	ELF binaries (Linux)
Kitsune (Hayden et al., 2012, 2014)	Updating software	Patch deployment and rollback	Semi-automated	C-based system
Ginseng (Neamtiu et al., 2006)	Updating software safely	Patch generation and deployment	Semi-automated	C programs

of the code, which runs simultaneously with the original code. Subsequent checks of old and new code behavior make it possible to verify and roll back broken patches if any discrepancies are detected. This approach requires operator assistance to define the potentially expected behavior discrepancies explicitly.

Several of the process-level patching approaches focus on the security and safety of the system after applying patches (Mugarza et al., 2020; Giuffrida et al., 2013; Pina and Hicks, 2016a). For instance, Giuffrida et al. (2013) have focused on safe and predictable updates of OS states. The proposed solution, Proteos, performs transactional live updates at the process level and requires programmer-provided state filters to limit updates to certain points in time only. Similarly, research is noticed on the consideration of the safety and security of a system while deploying patches (Mugarza et al., 2020) where a tool, Cetratus, is proposed for performing and verifying runtime patches in safety-critical systems. Pina and Hicks (2016a) have proposed TEDSUTO, which is focused specifically on patch verification steps. Old and new code behaviors are systematically tested, and discrepancies are identified as potential patch-related errors. This approach enables safe patch testing prior to applying it in a production environment. Similar to MVEDSUA (Pina et al., 2019), intended behavior changes must be explicitly annotated by developers.

Ramaswamy et al. have proposed Katana (Ramaswamy et al., 2010) which attempts to generate patches from source changes and safely apply them at run time. An interesting feature of the proposed solution is automatic safe update point detection. Specifically, the process execution stack is constantly monitored for this purpose, with suitable execution points being deduced on the fly. Stop-the-world technique is applied when a safe update point is encountered to replace the old code with the patched version. Authors, however, admit that their approach does not guarantee time bounds as a suitable execution point may not necessarily be reachable. Similar to Katana (Ramaswamy et al., 2010), Neamtiu proposed a tool, Ginseng (Neamtiu et al., 2006), that aims to integrate patch generation and deployment steps together. Function indirection and type wrapper are the key techniques used by Ginseng. Additional annotations are required from the developers to define safe update points.

Different from existing approaches, Bierman et al. (2003) have focused on formal foundation and conceptual reasoning about safe updating. Namely, an *update calculus* is proposed to enable formal reasoning about various code properties such as type safety or semantic correctness. Having a strong theoretical focus, this work does not tackle technical implementation-related issues such as specific programming languages or OS.

4.4. Container-level patching

Containerization is an emerging technology that has become widely popular in distributed computing applications. With the increasing adoption, various attacks, vulnerabilities and security challenges are noticed in distributed computing environments such as data centers and the cloud. Patching vulnerabilities in the container is not straightforward (Tunde-Onadele et al., 2020b,a; Sun et al., 2020; Ayres et al., 2021). Existing patching techniques often are not suitable for containers due to their short life and ephemeral nature (Tunde-Onadele et al., 2020b,a). Tunde-Onadele et al. have proposed lightweight on-demand patching techniques to patch containerized applications at runtime (Tunde-Onadele et al., 2020b,a) considering the resource constraints. Research in container-level patching techniques is noticed to enhance cloud security where blockchain-based integrity checkers are proposed to detect vulnerabilities in container images (Sun et al., 2020). The proposed approach further aids in generating and dynamically replacing vulnerable instances.

Recent studies have leveraged container virtualization and lightweight features to perform runtime patching in the automotive domain. For example, Ayres et al. (2021) have proposed to utilize the virtualization features of container technology in the Electronic Control Unit (ECU) of automotive vehicles to perform patching at runtime. The proposed approach is suitable for automotive functions which do not involve vehicle operation or safety and is instead focused on patching software related to autonomous driving or subsystems such as climate control. Table 4 shows that most of the container-level patching approaches are focused on automatically patching vulnerabilities, bugs or errors. Therefore, deploying patching at runtime is the main task of the studies categorized in container-level patching. However, the studies span server systems, container image security and automotive software.

Table 4 further summarizes the studies that proposed approaches for VM, Hypervisor and Kernel level patching.

Table 4
Container-level, VM-level, Hypervisor-level and Kernel-level patching.

Study	Goal	Task performed	Task type	Scope
Container				
SelfPatch (Tunde-Onadele et al., 2020a,b)	Patching vulnerabilities in container image	Patch deployment	Automated	Server
BlockChainPatch (Sun et al., 2020)	Patching vulnerability, malware (repository level)	Patch deployment	Automated	Container image security
ContainerECU (Ayres et al., 2021)	Patching vulnerabilities, bugs and errors	Patch deployment	Automated	Automotive software
VM				
Shadow Patching (Le et al., 2014)	Patching vulnerabilities	Patch deployment and testing	Automated	Cloud system and data center
vPatcher (Zhang et al., 2014)	Patching vulnerabilities	Patch deployment	Automated	Cloud platform and data center
Hypervisor				
Orthus (Zhang et al., 2019)	Patching vulnerabilities and feature updates	Patch deployment	Automated	Cloud system (Alibaba cloud)
HyperFresh (Doddamani et al., 2019)	Patching vulnerabilities and feature updates	Patch deployment	Automated	Cloud system and data center
Kernel				
Seamless (Siniavine and Goel, 2013)	Vulnerabilities and bugs	Patch deployment	Automated	Linux kernel
Kup (Kashyap et al., 2016)	Vulnerabilities, bugs, feature updates	Patch deployment and testing	Automated	Linux kernel
CRIU (Emelyanov and Kolyshkin, 2011)	Upgrading full kernel	Patch deployment	Semi-automated	Linux kernel

4.5. VM-level patching

VM-level patching granularity is conceptually similar to process-level, with the main difference relating to technical aspects. For instance, while replacing an existing process requires interacting with the OS, replacing a virtual machine would require interacting with the virtualization hypervisor.[5] We find only a few studies where a patch is performed at VM-level (Le et al., 2014; Zhang et al., 2014). Table 4 shows that VM-level studies mainly perform patch deployment automatically to patch vulnerabilities in cloud systems and data centers. Among them, Le et al. (2014) have proposed a shadow patching technique to reduce the downtime caused by patching VM running on the managed environment such as data centers. First, the proposed approach creates a replica of the VM that needs to be patched. Then, it deploys the patches in the replica VM, performs testing and makes necessary changes before applying the patch to the original VM. Finally, when the patches are applied to the original VM, the running application and data are merged from the replica VM to the original VM. This last step is similar to the VM teleport technique where current workloads are seamlessly transferred to the new VM instance.[6]

Different from the shadow patching approach, Zhang et al. (2014) have utilized VM introspective capabilities to monitor the behavior of VM using hypervisors to perform patching at runtime. Furthermore, they aim to perform data patching transparently in VM using hypervisor assistance. The proposed approach, vPatcher, is deployed outside the target guest VM to intercept and scan VM network traffic based on a vulnerability signature. This allows to filter out the network connections based on the vulnerability signatures.

4.6. Hypervisor-level patching

By hypervisor-level runtime patching, we consider studies that aim to replace the whole virtualization hypervisor without or minimally affecting the underlying virtual machines. Patching a hypervisor involves changing a large amount of dynamic data structures and is significantly complicated by the necessity to keep multiple underlying guest kernels intact and in sync with the hardware state. We observe a set of studies where hypervisors are patched at runtime without disrupting the running VMs (Doddamani et al., 2019; Zhang et al., 2019). Table 4 shows that hypervisor-level runtime patching is commonly applied to patch vulnerable cloud infrastructure without human involvement which mainly occurs in cloud and data center environments. In contrast to VM migration, hypervisor or VMM (Virtual Machine Manager) patching is typically performed in the same physical node and does not introduce significant network traffic. Thus, replacing or patching a hypervisor achieves better performance and lower downtime than VM live migration.

One technique to patch hypervisor at runtime is to copy the whole state of all active VMs from the running hypervisor to a new hypervisor (upgraded) (Zhang et al., 2019). The proposed techniques aim to seamlessly pass through devices to the new hypervisor automatically without losing any ongoing activities or operations state. Similarly, Doddamani et al. (2019) have proposed to transparently replace a hypervisor with a new one without disrupting or shutting down the VMs. The proposed approach, namely HyperFresh, introduces an intermediate layer to perform live hypervisor replacement. This layer maps the memory of the running VMs to the new locations belonging to the new hypervisor located in the same physical machine. These approaches focus on patching vulnerabilities and adding new features of cloud systems without interrupting the running VMs or shutting down the serves. While technically complicated, hypervisor-level patching is crucial in large commercial cloud deployments. Table 4 also shows that the studies on Hypervisor-level patching are mainly focused on deploying patches at runtime for patching vulnerabilities or updating features at cloud systems and data centers.

4.7. Kernel-level patching

We consider the patching solutions where a whole kernel is switched to a new kernel to apply all kinds of patches without

[5] Hypervisors support the creation and running of VMs, which is also known as Virtual Machine Monitor (VMM).

[6] https://docs.oracle.com/en/virtualization/virtualbox/6.0/admin/teleporting.html

having to restart the kernel (Kashyap et al., 2016) as Kernel-level runtime patching. We noticed a few studies (Kashyap et al., 2016; Siniavine and Goel, 2013; Emelyanov and Kolyshkin, 2011) that have performed runtime patching at kernel-level. Kernel-level patching is executed through the use of kernel space and user space checkpoint and restart mechanism (Kashyap et al., 2016; Siniavine and Goel, 2013; Emelyanov and Kolyshkin, 2011). Table 4 also shows that few studies have proposed to replace the whole Kernel while patching at runtime. Unsurprisingly, from Table 4 we can see that the kernel-oriented solutions mainly target Linux, most likely due to the source code availability. The goal of these studies includes patching vulnerabilities, fixing bugs, updating features and upgrading an entire kernel.

In place kernel switch is a popular technique to update OS that heavily modifies various subcomponents of the kernel (Siniavine and Goel, 2013). The proposed approach mainly targets restoring the applications entirely in kernel space. After applying patches to the Kernel, it is essential to ensure that the changes do not break or corrupt the existing functionalities and applications. Thus, the testing process of kernel-level patching is costly and tedious (Xu et al., 2020). Kashyap et al. (2016) have proposed to perform runtime kernel patching without modifying the kernel source code. They have adopted binary patching techniques, userspace checkpoint and restore, and optimized in-place Kernel switching to perform the kernel patching. Alternatively, in order to simplify kernel updating, resource (such as running processes and containers) handover assistance can be utilized (Emelyanov and Kolyshkin, 2011). For instance, a widely used tool, CRIU[7] makes it possible to freeze critical processes, replace the whole old Kernel with a patched/new version and restore the frozen processes. While CRIU was initially designed for live container migration, further project development has enabled the runtime kernel upgrading scenario.

In summary, we observe that function-level patching techniques are widely proposed in the literature, followed by instruction-level and process-level patching. The key reason we notice is the granularity of the patch; with function and instruction level patching, there require more minor changes in the resources as shown in Fig. 3. On the other hand, we notice little research in patching heavy loaded resources such as VM, container, hypervisor and Kernel due to the complexity and dependency of the resources and tasks with other resources.

5. Patch deployment strategies, workflow and capabilities

This section provides an overview of the patch strategies the reviewed patching solutions employed, along with patch deployment workflows commonly implemented in practice. *State transformation* and*co-exists & decay* are two identified patch strategies that are discussed in detail in Section 5.1. Despite the high variability of technical implementations, a detailed look into the patch deployment steps in Fig. 1 reveals several commonalities between the existing runtime patching solutions. Individual patch deployment steps vary in terms of patch strategies which are discussed in Section 5.2. Crucial technical capabilities exhibited by the reviewed solutions are discussed in Section 5.3.

5.1. Runtime patching strategies

The analysis of existing runtime patching approaches reveals two main patch deployment strategies (i) State transformation and (ii) Co-exist & decay. These patch strategies refer to the conceptual models underpinning the patch handling steps. Here, we discuss the general goals and trade-offs of the identified patching strategies.

[7] https://www.criu.org/Main_Page

5.1.1. State transformation

State transformation attempts to modify the currently running code to fix or remove the unwanted behavior. State transformation line of thought aims to convert the old code along with all the necessary resources, data or states to match the expected behavior of the patched code as shown in Fig. 4(a). This approach ensures that the patched code is effective immediately as soon as the state or data transformation is completed (Chen et al., 2018; Kashyap et al., 2016; Siniavine and Goel, 2013). State transformation strategy can be implemented in cooperative and uncooperative manners.

Cooperative patching relies on some degree of assistance from the old code. Usual assistance avenues include auxiliary compiler- or developer-provided code or annotations that simplifies determining quiescent execution state and/ or data transformation procedures. This, however, requires designing the application to provide such cooperation from the grounds up and is not suitable for legacy systems. Hence, cooperative state transformation is mainly viewed as part of new development practices and tooling.

Uncooperative patching implies that the old code is completely unaware of possible updates, and the whole patching burden falls on the patching subsystem. In particular, this means that the patching code has to possess intimate knowledge of the internals of the old code. Existing resources must be located, detached from the old code and converted. Depending on the type of software, this would typically mean parsing and modifying old code RAM contents. While this type of patching is technically more challenging than the cooperative one, the main advantage is the broader applicability. Uncooperative patches can be potentially applied to a wider set of existing legacy software systems.

The main obstacles encountered by state transformation strategies are that the transformation process itself is not instantaneous and not applicable at any moment in time. Modifying arbitrary data bytes in memory may negatively impact current execution threads that actively use modified data. Furthermore, uncoordinated data modifications may introduce inconsistencies between different parts of code, which can lead to unforeseen consequences, side effects and behavior. Thus, state transformation strategy involves waiting for a less active (quiescent) state to be reached for software or code execution. Reaching a quiescent state allows for safely transforming the necessary memory objects. However, this is not straightforward as some code may include a long-living function that rarely (if ever) ceases execution. Furthermore, due to external factors, highly-loaded applications may never reach a quiescent (safe-to-update) state. In addition, as state transformation is not instantaneous, any concurrent access to the memory object being transformed must be prevented. Therefore, code execution is typically briefly suspended while the transformation is conducted. Note that while such execution suspension is technically a service interruption, short interruptions are assumed to go unnoticed by external observers. State transformation attempts to modify the currently running code to fix or remove the unwanted behavior. State transformation line of thought aims to convert the old code along with all the necessary resources, data or states to match the expected behavior of the patched code as shown in Fig. 4(a). This approach ensures that the patched code is effective immediately as soon as the state or data transformation is completed (Chen et al., 2018; Kashyap et al., 2016; Siniavine and Goel, 2013).

5.1.2. Co-exist & decay

Arbitrary state transformations may be either prohibitively computationally expensive or impossible for complicated software commonly used in practice (Gupta et al., 1996). Thus, an alternative strategy *Co-exist & decay* is proposed that attempts to side-step the transformation-related issues by exploiting the

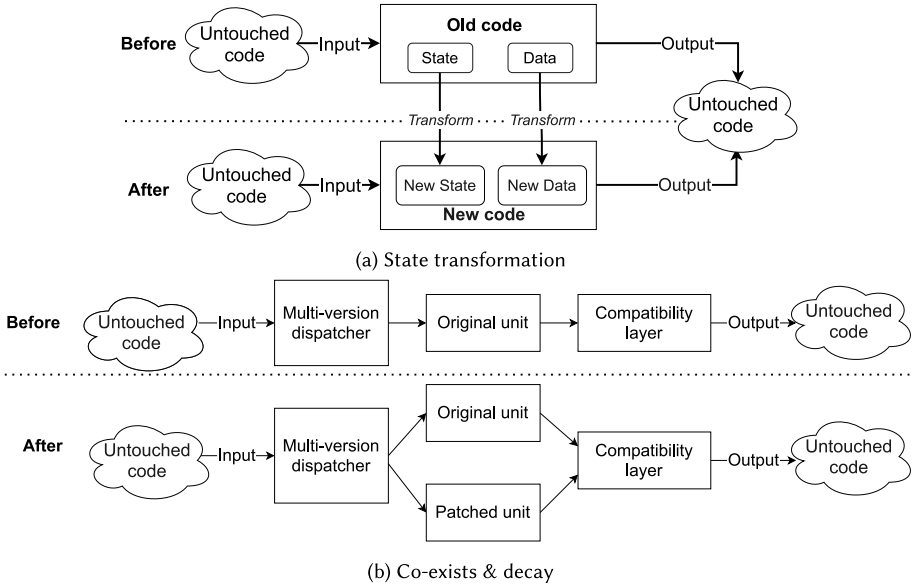


Fig. 4. Conceptual view of patch strategy (a) co-exist & decay and (b) state transformation before and after applying a patch.

transaction-based nature common to modern software services. Depending on the granularity in consideration, such transactions can vary from individual CPU instructions and functions to higher-order logically independent user-software interactions. Fig. 4(b) shows a high-level overview of co-exist & decay approach. Certain network-based software and web applications in particular are prime examples, where users periodically establish independent sessions or send network requests based on their needs.

Coexist & decay strategy attempts to completely avoid any interruptions at the expense of a potentially more prolonged code activation wait period. Using various forms of indirection in code execution dispatching mechanisms enables having multiple code paths available. In coexist & decay, old and new codes coexist in the system simultaneously. Any new external interactions go through the new code execution path, while currently, in-flight interactions are processed by the old code. As already started interactions finish off, old code execution paths are decayed (i.e., removed with associated memory objects disposal). For instance, web-based applications benefit from the stateless nature of the underlying HTTP protocol, with the patches being possible to apply between requests. In addition, the time intervals between separate requests provide natural, safe update points. These features make web applications a perfect fit for coexist & decay solutions.

In addition to enabling the use of multiple concurrent versions of code, another important role of the dispatcher is to perform actual code path selection at run time. Significant changes implemented by complex patches may introduce some incompatibilities, primarily with external code. For instance, patches that alter network protocols on the server side may render remote network clients (that are not yet updated accordingly) incompatible with the new protocol version. Thus, the compatibility layer of the dispatcher, as shown in Fig. 4(b) needs to differentiate between transactions/interactions on the basis of code version compatibility. Further, throughout execution, once all existing interactions for a given version are finished, the associated code path can be decayed.

5.2. Runtime patching deployment workflow

Deploying or applying a patch at runtime consists of several preparatory steps followed by actual patch code activation steps. The preparatory steps of patch deployment are common for all patch approaches, while patch activation steps depend on the employed/ selected strategy (i.e., state transformation and co-exist & decay). As can be seen from Fig. 5, allocating memory, loading patch into RAM and configuring patch are common steps. Supported software environments such as OS, programming language and machine architecture greatly affect how each of these steps are implemented by a given patching solution (Zhou et al., 2020; Chen et al., 2013; Xu et al., 2020; Tunde-Onadele et al., 2020b). Note that Fig. 5 outlines the conceptual steps that do not depend on the particular granularity chosen. The outlined steps need to be taken regardless of the granularity selected; however, the actual implementation may differ. The type of resources to transform or dispatch depends on the patch granularity and nature (Fig. 3). For instance, patching a whole VM compared to patching an individual instruction typically requires transforming more resources (e.g., network sockets, file handles, hypervisor objects, etc.).

Allocating memory to accommodate the patched code in RAM is a necessary step for all solutions except those that perform limited instruction overwriting in-place (such as Chamith et al. (2016)). *Loading the patched code into RAM* is necessary regardless of the technical differences in the implementation. Lastly, *configuration of a patch* may include sub-tasks such as decompression, decryption or digital signature verification than just copying patch bytes into memory. These sub tasks are optional and mainly implemented to increase performance or reliability. In some cases, the three preparatory steps may become completely unnecessary, complicated, impossible or require external intervention. For example, resource-limited embedded IoT devices may not have enough unused memory to fit the patched code into RAM, leading to the necessity to completely erase and re-flash the device through external means/devices.

Fig. 5 shows further steps of *patch activation* that depend on the general patching strategy employed. In efforts to minimize any potential system or service downtime, runtime patching solutions typically attempt to perform as many tasks as possible prior to activation of the patched code. For instance, *concurrent state transformation* may occur with execution of old code wherever possible. As a consequence, once the *quiescent state* is reached, the number of actual code activation steps would be reduced and thus take less time to complete. The *patch code activation* step is

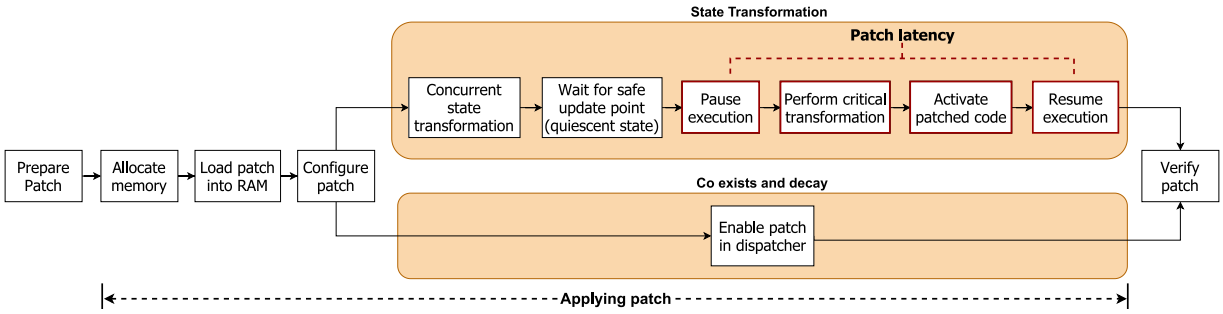


Fig. 5. Workflow of deploying patching at runtime.

time-critical as the *patch latency* period between code execution *pausing* and *resuming* should be kept minimal. Thus, only *critical state transformation* (that were not possible outside of quiescent state) should be conducted while the code execution is paused. While not having issues with quiescence, the patch enabling step in dispatcher used by coexist & decay (as shown in Fig. 5) must deal with non-instantaneous patch activation instead.

In both cases, the potential downside is that the old (potentially vulnerable) code would operate for some time, while the preparatory steps are taken, the quiescent state is not yet reached or new interaction starts. Having the vulnerable code still active, might pose a significant security risk for large-scale patches (i.e., VM- or kernel-sized), leading to the necessity to completely stop any system operation temporarily. The overall state transformation can then be performed in a safe manner. Such execution suspension would essentially be equivalent to traditional offline patching in terms of user experience disruption.

5.3. Patch implementation capabilities

Several generic capabilities implemented by existing patching systems have been identified throughout this study. In line with the presented patching workflow of Fig. 5, we identify four main capabilities (see Fig. 2) — (i) memory management, (ii) resource transformation, (iii) safe execution state determination and (iv) execution path dispatching. The implementation details of the runtime patching systems exhibit these capabilities differently in the sense of limited applicability to certain use cases. For instance, in most of the reviewed solutions, these capabilities are limited to specific programming languages, OSes and environments. Table 5 summarizes an overview of the patch implementation capabilities of different patch approaches along with execution or operational environment. It also shows the advantages or quality requirements promised to be fulfilled by a particular study.

5.3.1. Memory management

In contrast to traditional offline patching, which deals with file-level abstraction, one of the key tasks that a runtime patching system must solve is related to memory management. This includes finding necessary RAM locations, allocating memory for new code, relocating memory objects and so on (Chen et al., 2013; Salls et al., 2017; Doddamani et al., 2019; Zhou et al., 2020). *Determining RAM addresses* is typically required for lower-level patch granularities such as instruction- or function-level. More specifically, locating the old code address in memory is required to disable or overwrite vulnerable portions with the patched version. In some cases, such as when the length of the patched code is longer than the original, a new (free) memory region needs to be allocated and used.

One of the popular solutions in these cases is the use of trampolines, where a few instructions of the original code are replaced to point to the patch code that is stored in the newly-allocated memory space (Zhou et al., 2020; Salls et al., 2017; Xu et al., 2020). However, such trampolines do not address internal jumps and breaches to intermediate levels. This leads to the use of function trampolines during function-level patching. For example, Salls et al. (2017) have proposed Piston that replaces individual functions. Piston adopts trampolines that are essentially direct jump instructions at the beginning of the old functions to perform function replacement. In addition, implementing trampolines requires checking if the trampoline jump instructions fit entirely within the function body. This is partially mitigated in modern operating systems and compilers by aligning functions to 8-byte address boundaries, which should be enough for direct jump instructions.

Some patching systems attempt to simplify or completely eliminate the memory address locations task. For instance, relying on APIs exposed by existing dispatching mechanisms in the code, the patching system may not even need to find a concrete memory address to modify. Note that using OS-specific features such as Address Space Layout Randomization (ASLR) technology may significantly complicate address calculations (Gu and Lin, 2016). While technically involved and situation-specific, locating memory addresses has known OS-specific solutions (Gu and Lin, 2016).

Despite being highly platform-dependent, allocating additional memory regions for new code and data objects is relatively simple compared to specific RAM addresses finding tasks. The main problem is the amount of RAM to allocate. This might not be an issue for modern desktops and servers but could pose challenges in embedded and resource-constrained devices.

5.3.2. Resource transformation

Due to the requirement of continuous operation, a crucial capability of a patching system is to achieve compatibility between currently occurring activities and the expected new behavior. Existing resources (e.g., runtime data objects) may need to be converted to comply with the new code expectations while maintaining compatibility with the existing external users. Note that, particularly in uncooperative patching scenarios, the resources to be transformed must first be determined and located. While locating known objects in RAM can be simplified by utilizing compiler or dynamic linker information, determining the actual list of objects to transform is more complicated. Furthermore, the resources that need to be transformed to be compatible with the new code depend on the actual change introduced by the patch. In contrast, cooperative solutions may rely on continuous state tracking to aid the new code when necessary. Similar state tracking approaches relying on safe checkpoints are commonly used in intermittent computing (Lucia et al., 2017).

Gupta et al. (1996) have shown that arbitrary program state transformations are not possible in the general case. Thus, existing solutions are typically limited to particular programming

Table 5
Runtime patching approaches with respective capabilities, granularities and quality requirements. The studies are mapped with four supported capabilities: Memory management (MM), Resource transformation (RT), Safe Execution State Transformation (SESD), and Execution Path Dispatching (EPD).

Study	Execution or operational environment	MM	RT	SESD	EPD	Granularity	Focus (Quality Requirement)
D-linking (Hicks and Nettles, 2005)	TALx86	–	–	✓	✓	Instruction	Flexibility, Robustness, Ease of Use, Effectiveness (low overhead)
Kshot (Zhou et al., 2020)	X86 SMM, intel SGX	✓	✓	–	–	Function	Effectiveness (low overhead), Trustworthiness
Safestack (Chen et al., 2013)	Linux	✓	–	✓	✓	Instruction	Efficiency, Safely, Scalability
Instaguard (Chen et al., 2018)	Android	✓	–	✓	✓	Instruction	Efficiency
Replus (Chen et al., 2015)	Linux	✓	–	✓	✓	Instruction, Function	Efficiency, Lightweight, Practicality
DUSC (Orso et al., 2002)	Java runtime system	–	✓	–	–	Function	Effectiveness (low overhead)
Vulmet (Xu et al., 2020)	Android	–	–	✓	–	Instruction	Correctness, Robustness, Efficiency
Seamless (Siniavine and Goel, 2013)	Linux	–	✓	✓	–	Kernel	Effectiveness, Reliability
Kup (Kashyap et al., 2016)	Linux	–	✓	✓	–	Kernel	Effectiveness, Reliability
Cure (Zhao et al., 2016)	x86	–	✓	✓	–	Function	Effectiveness, Safety
Piston (Salls et al., 2017)	x86	✓	–	–	✓	Function	Effectiveness, Persistence
DynSec (Payer et al., 2013)	x86	✓	–	✓	✓	Instruction	Correctness, Effectiveness (low overhead), Efficiency
Gitar (Ruckebusch et al., 2016a)	Contiki, TinyOS	–	–	–	✓	Function	Efficiency, Sustainability, Maintainability
HotAsap (Payer and Gross, 2013)	Apache web server	✓	–	–	–	Instruction	Correctness, Effectiveness (low overhead), Efficiency
Karma (Chen et al., 2017)	Android	✓	–	–	–	Instruction	Adaptiveness, Safety, Effectiveness
Multiverse (Rommel et al., 2019a)	Linux, cPython, musl	✓	–	–	✓	Function	Efficiency, Ease of use, Flexibility, Portability
C-Multiverse (Rothberg et al., 2016)	Linux, cPython, musl	✓	–	–	✓	Function	Efficiency, Ease of use, Flexibility, Portability
BinPatch (Hu et al., 2019)	Linux, x86	✓	–	–	–	Function	Efficiency, Persistence
ShawdowPatching (Le et al., 2014)	Virtual	–	✓	–	–	VM	Effectiveness
Proteos (Giuffrida et al., 2013)	x86 Proteos	✓	✓	✓	–	Process	Effectiveness, Scalability, Reliability, Security
Waitfree (Rommel et al., 2019b)	x86, Linux	–	–	✓	✓	Process	Low overhead
Vpatcher (Zhang et al., 2014)	Virtual	–	–	–	✓	VM	Correctness, Efficiency
Orthus (Zhang et al., 2019)	Virtual	–	✓	–	–	Hypervisor	Efficiency, Effectiveness, Scalability
HyperFresh (Doddamani et al., 2019)	Virtual	✓	✓	–	–	Hypervisor	Effectiveness, Low overhead
ContainerECU (Ayres et al., 2021)	Electronic Control Unit (ECU)	–	–	✓	–	Container	Efficiency, Effectiveness (low overhead), Scalability
Cetratus (Mugarza et al., 2020)	x86, PowerPC	–	–	–	✓	Process	Stability
MVEDSUA (Pina et al., 2019)	C program	–	✓	–	✓	Function	Availability, low latency, effectiveness
Tedsuto Pina and Hicks (2016b)	Java program	–	✓	✓	–	Process	Effectiveness, Efficiency
Kitsune (Hayden et al., 2012, 2014)	C-program	–	✓	✓	–	Process	Effectiveness (no overhead), Efficiency
Ginseng (Neamtiu et al., 2006)	C program	–	✓	✓	–	Process	Scalability, Effectiveness (no overhead), Efficiency
Polus (Chen et al., 2007)	x86	✓	✓	–	✓	Function	Effectiveness (low overhead), Efficiency

(*continued on next page*)

Table 5 (*continued*).

Study	Execution or operational environment	MM	RT	SESD	EPD	Granularity	Focus (Quality Requirement)
Ksplice (Arnold and Kaashoek, 2009)	Linux kernel	✓	✓	✓	–	Function	Efficiency
Appwrapper (Lee et al., 2020)	Java program	–	–	–	✓	Function	Efficiency, Effectiveness
OSSPatcher (Duan et al., 2019)	C/C++ program	–	–	–	✓	Function	Efficiency, Feasibility, Variability
HotPatcher (Jeong et al., 2017)	ARM, x86_64	✓	–	–	✓	Function	Effectiveness (low cost and overhead), Reliability

languages and types of transformations only. For instance, only certain types of changes, such as adding a field to a class or changing the number of function parameters, may be supported by some solutions (Lee et al., 2020). Naturally, different technical solutions would be used depending on the level of patch granularity supported and execution environment.

In cases when most of the compatibility is expected to be retained, simplified resource detachment from old code and reattaching to new code can be enough. For instance, new OS kernels are typically assumed to be compatible with the existing applications. Therefore, as updating the whole OS kernel is not expected to affect running processes, *checkpoint and restore*-type approaches can be employed in practice (Kashyap et al., 2016; Siniavine and Goel, 2013). These approaches essentially transplant resources from old code to new code. Resource transition from the old code to the new code is referred to as handover if the old code provides some transformation assistance. Conversely, if the new code needs to transform the resource without any old code participation, the operation is referred to as a takeover.

The resources transferred could refer to running OS processes (Kashyap et al., 2016; Emelyanov and Kolyshkin, 2011) or memory pages of a VM running in a hypervisor (Doddamani et al., 2019). A significant advantage of resource transplanting is the lack of necessity to modify resources themselves as long as a high degree of compatibility is retained between patches. The most significant downside is the lack of control over external dependencies. For instance, restoring a previously running process to run under a new kernel cannot guarantee that network sockets used by the process would still be open on the remote side. Note that some complicated patches may introduce significant changes that are breaking compatibility with external entities outside of control or area of responsibility of the patch. Notably, introducing incompatible changes to the networking code of a server may render all legacy network clients incompatible, thus causing large-scale severe service disruptions.

5.3.3. Safe execution state determination

During runtime patch deployment, extra care must be taken to prevent external execution threads from executing or accessing code and data fragments that are being modified. A naïve approach of straightforwardly suspending all other threads (known as stop-the-world) for the duration of the patching process is not adequate as resuming those threads after patching might lead to inconsistencies. Furthermore, suspending running threads may cause noticeable service interruption for more prolonged (i.e., large-scale) patching procedures. Therefore, runtime patching solutions aim to determine time slots that are safe for thread suspension and suspend execution threads for the shortest time possible.

A safe patching time slot refers to a point in software execution where the patched code or data is not in use. For example, if rewriting instructions comprising a function are conducted when the function is being executed, half-changed instructions may be executed, leading to inconsistent behavior. Depending on the execution environment, various safe timing detection algorithms may be implemented in practice.

Two main possibilities are to detect when a certain execution condition is met or rely on existing code assistance when possible. Detection is typically achieved through memory access monitoring and breakpoints mechanism. Specifically, a monitoring stack is commonly used in practice to determine whether the given function is currently being executed (Chen et al., 2015). Naturally, relying on existing code assistance is only possible when such assistance facilities are part of the code design and not for legacy software. Automated patch code analysis may reveal critical code points suitable for safe patching and set corresponding breakpoints. Once such breakpoints are triggered, the patching deployment can commence safely. Alternatively, semi-automated patching solutions relying on safe time detection assistance use function hooks to allow notifying the patching system when a safe execution point is reached (Zhao et al., 2016). In some sense, safe point detection is offloaded to the original software developers and not directly implemented by the patching system.

Interestingly, Intaguard (Chen et al., 2018) attempts to fix the software behavior only when vulnerability exploitation is detected rather than patching code in advance. Instaguard waits for program execution to reach a vulnerable instruction, then after intercepting the execution, corrects the program behavior and returns execution to the original code. In some sense, rather than looking for *safe* patch time slot only a *necessary* moment is being detected in case of Instaguard (Chen et al., 2018). Lastly, some patching solutions side-step the necessity to detect safe time points by following the Co-exist& Decay strategy. In these situations, patch safety is achieved by isolating different user requests or sessions not to impact each other to avoid potential inconsistencies.

5.3.4. Execution path dispatch

Code execution path dispatching is commonly achieved through various *indirection* layers. This indirection could refer to either code fragments or individual data objects. Generally, **indirection** is commonly used to achieve loose coupling between different entities. Fig. 6 illustrates several indirection layers frequently used in modern software systems. Indirection layers such as shown in Fig. 6 serve multiple purposes, including user convenience. For instance, remembering human-readable DNS names is easier for humans compared to numeric IP addresses. In the context of software patching and updating, the loose coupling achieved through indirection simplifies replacing old entities (e.g., function and instruction) with the patched ones. Specifically, updating the mapping in the indirection layer enables a seamless transition to the new entities. For example, updating DNS records to point to new IP addresses, changing symbolic link destination or redefining CPU instruction (in terms of μOPs) ensures that future usage would point to the updated entity.

Merely replacing a buggy entity with the fixed one is, however, not enough as old entities might still be in active use by existing execution threads. In the case of long-living usage transactions (i.e., network sessions), the old entities may not be released for prolonged periods. Therefore, various dispatching mechanisms

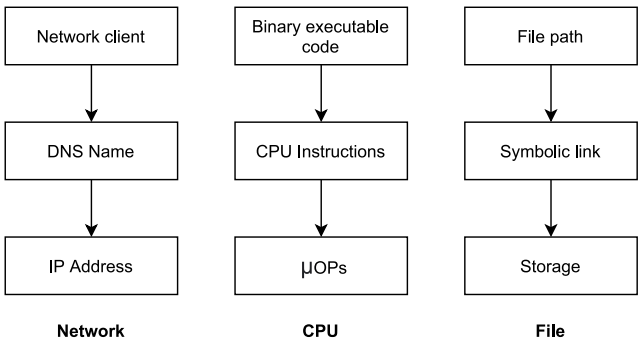


Fig. 6. Examples of network, CPU and file-level indirection. μOps refers to micro operations of CPU.

are implemented *in conjunction with indirection layers* to control which of the entities would be used at a given time (Rommel et al., 2019a; Rothberg et al., 2016). Such dispatching mechanisms are the core of the co-exist & decay strategy, with the exact definitions of *entity* and dispatching algorithms used widely varying depending on the patching granularity supported.

Explicit version numbers are commonly used for dispatching[8,9] Such explicit versioning ensures compatibility between different copies of the co-existing code or data and external users. Due to the associated performance and memory overheads, multi-version dispatching may be used as a temporary measure while a developer is working on a longer-term solution (Rommel et al., 2019a; Rothberg et al., 2016). In addition, to minimize at least the memory overhead, the gradual decay (i.e., cleanup) of old code and resource may be implemented as a feature of some patching solutions.

6. Runtime patching responsible entities

The entities and their roles involved in the runtime patching process are another important aspect of the patching workflow that is not commonly explicitly discussed. We identify three key parties involved in the typical patch life cycle, which are (i) the original software vendor, (ii) software end-user/consumer and (iii) third party patch system developer. Fig. 7 depicts an overview of three different scenarios of patch deployment with respect to different parties and their corresponding responsibilities. Consider that the same entity could fulfill all three roles in simple cases. For instance, for an in-house developed software used within company boundaries only, the company develops software (along with associated patches), uses the software and provides further maintenance through implementing patches as required. Note that we explicitly focus on patch deployment rather than patch development or automated generation responsibilities. However, the framework Replus has considered both developer and consumer perspectives of patching and divided the patching duties between developers and consumers (Chen et al., 2015). Table 6 provides a mapping of various granularity with patch responsibility proposed in the literature review.

6.1. Vendor-assisted patching

In vendor-assisted patching scenarios, the software is designed with runtime patching support in mind, with end-users having little to no involvement in the patching process (Rommel et al., 2019a; Rothberg et al., 2016; Ruckebusch et al., 2016b; Mugarza et al., 2020). With modern software commonly enabling automated patching by default, users at most have an option to apply or skip a given patch. Original software vendors have deep knowledge of how the software operates, which (in principle) enables developing complex and safe patches. Also, having access to the software sources opens up several avenues not available for end-users or third parties in closed-source software. For instance, runtime patches could be automatically generated based on the source code changes conducted by the software developers. Having control over software development, deployment and further management enables designing tightly integrated patching solutions (Giuffrida et al., 2013). Patches can be developed, delivered to end-users and automatically deployed by the vendor as shown in Fig. 7(a), which significantly reduces the burden of management on the end-users.

Two potential downsides of vendor-assisted patching are low applicability for legacy systems and lack of flexibility from an end-user perspective. Legacy systems lacking a run-time patching subsystem would fall victim to potential vulnerabilities. Completely redesigning such legacy systems might be prohibitively expensive or impractical. End users would also be entirely dependent on the willingness of the software vendors to fix a given issue. Lower priority or site-specific problems may not receive enough attention from the developers, causing a lack of patches addressing some of the end-user needs.

6.2. Consumer-assisted patching

In some cases, software consumers may opt to manage patch deployment for the software they use, which we refer to as consumer-assisted patching. Three possible reasons include a higher level of control, more flexibility and lack of support from the original software vendor. At the expense of increased labor required, implementing own patching strategy essentially allows end-users to gain a higher level of control over the software system as shown in Fig. 7(b). Consequently, fine-grained or user-specific patches can be deployed by the consumers based on their current and ever-changing requirements (Salls et al., 2017; Hayden et al., 2012, 2014). Also, implementing own patching solutions may be the only option for legacy systems that do not include runtime patching support or are no longer maintained.

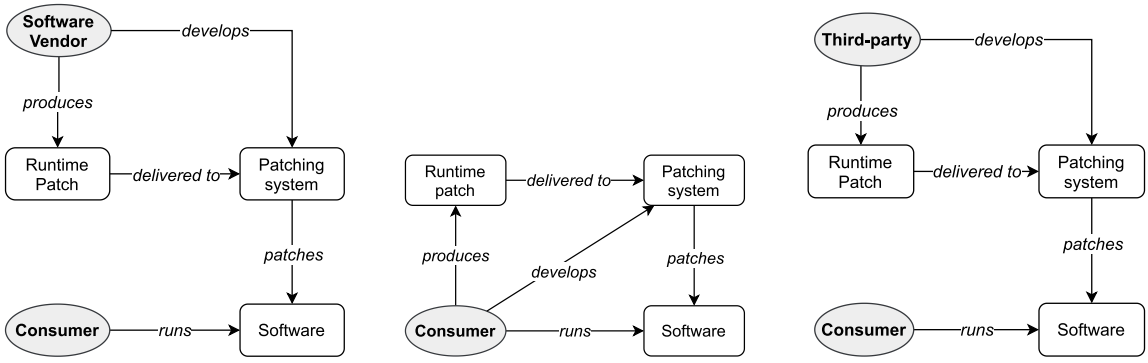
However, achieving the desired flexibility may not be easy for the end-users as they lack intimate knowledge of data structures and flow used within the software. Due to the complexities in foreseeing the potential side-effects of the patch-induced changes, applying potentially inconsistent/faulty patches poses significant risks. The complexity of predicting potential patch failures or discrepancies is higher for closed-source software while being somewhat more manageable for fully open-source software. In a subset of situations, end-users might limit their efforts to converting traditional source- or binary-level patches into a dynamic form rather than developing their patches from scratch. In other words, end-users might focus on developing a runtime patching system to deploy the traditional vendor-produced offline patches in a dynamic manner.

6.3. Third-party-assisted patching

Third-party runtime patching systems refer to solutions that are applicable in a wider set of software environments and represent a practical trade-off between vendor and consumer efforts. Not possessing the knowledge of inner software details and lacking information on individual end-user needs, third parties aim to achieve some level of generality in terms of applicability in different contexts such as specific programming languages, OS or

[8] https://docs.microsoft.com/en-us/aspnet/core/grpc/versioning?view=aspnetcore-6.0

[9] https://github.com/grpc/grpc/blob/master/doc/versioning.md



(a) Vendor assisted (b) Consumer assisted (c) Third party assisted

Fig. 7. System overview of runtime patching system with different responsible entities — (a) software vendors, (b) consumers and (c) third parties. Third parties refer to patch system developers who are not vendor or consumer. The oval shapes in the diagram represent human entities and the square shapes represent software entities.

Table 6
Analysis of existing patching solution with patch responsibility.

Granularity	Vendor	Consumer	Third party
Instruction	–	DSU (Hicks and Nettles, 2005)	DSU (Hicks and Nettles, 2005), Safestack (Chen et al., 2013), InstaGuard (Chen et al., 2018), Replus (Chen et al., 2015), Vulmet (Xu et al., 2020), Wordpatch (Chamith et al., 2016)
Function	Multiverse (Rommel et al., 2019a), CompMultiver (Rothberg et al., 2016), Gitar (Ruckebusch et al., 2016a)	Ksplice (Arnold and Kaashoek, 2009), Piston (Salls et al., 2017), UpdateCalculus (Bierman et al., 2003)	Replus (Chen et al., 2015), Polus (Chen et al., 2007), Cure (Zhao et al., 2016), Appwrapper (Lee et al., 2020), OSSPatcher (Duan et al., 2019), HotPatcher (Jeong et al., 2017)
Process	Katana (Ramaswamy et al., 2010), Proteos (Rommel et al., 2019b), MVEDSUA (Pina et al., 2019), Ginseng (Neamtiu et al., 2006)	Cetratus (Mugarza et al., 2020), Proteos (Giuffrida et al., 2013)	Tedsuto Pina and Hicks (2016a), Kitsune (Hayden et al., 2012, 2014)
VM	–	–	ShadowPatching (Le et al., 2014), Vpatcher (Zhang et al., 2014)
Hypervisor	Orthus (Zhang et al., 2019), HyperFresh (Doddamani et al., 2019)	–	–
Container	–	–	Opatch (Tunde-Onadele et al., 2020b), SelfPatch (Tunde-Onadele et al., 2020a)
Kernel	–	–	Seamless (Siniavine and Goel, 2013), Kup (Kashyap et al., 2016)

certain patch granularity. For instance, some solutions support solely Java- or C-based applications (Orso et al., 2002; Payer and Gross, 2013; Hayden et al., 2014), while others focus on specific OS environments (Giuffrida et al., 2013; Chen et al., 2018).

The third-party-assisted patching approaches do not expect assistance from the old code while still simplifying patch deployment. One common theme within third-party efforts is to focus on converting existing source-based vendor-developed offline patches to be applied at run time by end-users. For this purpose, third-party-assisted software updating implements an external entity that would replace old code with the new code while preventing potential service interruptions. For instance, the proposed approach by Chen et al. (2013) requires the installation of a runtime patch applicator in the online production system. Fig. 7(c) shows a high-level overview of third-party-assisted patching. Most of the research efforts we have studied are geared towards third-party assisted patching (Xu et al., 2020; Chen et al., 2018; Lee et al., 2020; Duan et al., 2019; Chen et al., 2007; Zhao et al., 2016; Pina and Hicks, 2016b; Zhang et al., 2019; Tunde-Onadele et al., 2020a; Kashyap et al., 2016).

7. Discussion

The high performance and memory overheads primarily define the runtime patching viability as opposed to traditional offline patching. These overheads are caused by the sheer size of the execution state inherent to modern complex software and difficulties in determining safe update points. In mission-critical domains such as medical, military or rescue operations, it is essential to maintain uninterrupted service operations. Highly critical systems with plentiful resources can tolerate high performance or memory overheads common for runtime patching systems. However, the resource-constrained systems such as embedded IoT devices may require more efficient patching solutions with lower performance or memory overheads. Our analysis of the reviewed studies hints that vendor-provided patches have more potential to reduce the associated overheads. The intimate knowledge of the software internals along with the collaborative nature of software, simplify many of the patching steps. Collaborative nature can refer to requesting controlled code execution suspension instead of abrupt code interruption or lengthy waiting of quiescent state. Similarly, the existing old code can collaborate in providing the list of resources in use rather than requiring complicated manual resource tracking.

According to our proposed taxonomy, we also conclude that runtime patching is dominant in open-source ecosystems, with the majority of patching frameworks designed to be used by independent third parties as opposed to software vendors and consumers directly (as shown in Table 6). This is likely explained by software vendors not paying enough attention to runtime

patching functionality. An intermediate middle-man role seems to be forming, focused on service continuity. Cloud infrastructures are a typical example where service continuity is not a separate solution exposed to end users by software vendors but is instead integrated into the cloud offering by the cloud provider. Access to source code significantly simplifies the understanding of application internals, enabling the development of efficient runtime patching solutions suitable for a given application. Not limiting a patching solution to particular vendors or end-users also contributes to a patching system's popularity due to its broader applicability.

We observe a common commercialization trend in runtime patching solutions where rather than monetizing the patching system itself, actual patches are provided as part of paid subscription. For example, the existing commercial solutions such as KernelCare by TuxCare,[10] Ksplice by Oracle[11] and kpatch by Red Hat[12] provide patches that are possible to apply at run time. Customers essentially pay for adapting and supporting regular source-level changes to runtime patch format. Note that some types of changes, such as semantic structure modifications or specific system functions replacement, may be explicitly unsupported by certain patching solutions.[13,14]

We further observe that despite these successful commercial solutions focusing on Linux kernel patching, the actual granularity used internally is not necessarily kernel-level but can be implemented at instruction or function levels (Tables 1–4). Notably, we categorize Ksplice into function-level granularity. Consequently, depending on the patch design choices, different capabilities, as shown in Table 5, may need to be implemented. For instance, Ksplice opted not to implement execution code path dispatching capability while supporting the other three patch implementation capabilities identified by our taxonomy.

Closed-source commercial runtime patching solutions are predominantly developed, controlled and managed by the corresponding software vendors. This is because only vendors have access to software source code, which significantly simplifies patch development and preparation. Consequently, such vendor-curated patching solutions are tightly integrated into software and not provided separately for a wider audience. For instance, Microsoft employs a run-time patching solution to minimize Azure SQL Database engine service disruptions during patching.[15] According to their statistics, over 80% of SQL bugs can be remediated using this patching system. As per our taxonomy, this approach is a function-level coexist and decay solution. Despite being closely tied to a particular software application, a similar compiler-assisted approach can be used for other Visual C-based applications due to the ubiquity of patch loading and function redirection steps. Another commercial solution we found, namely MuleSoft CloudHub platform,[16] primarily focuses on maintaining continuous cloud service. Due to the closed nature, the provided patching solution description is scarce on implementation details; however, according to our taxonomy falls under process-level coexist and decay solutions.

Note that while some dynamic programming languages such as Python or JavaScript provide various dynamic code loading mechanisms, they do not directly map to actual patch contents. A developer must still prepare the code that will conduct necessary state transformations by writing a piece of code that will be loaded and applied. Nevertheless, further development of dynamic code handling is a promising path towards efficient runtime patching system implementation.

8. Open challenges and future research directions in runtime patching

A number of observations related to gaps in the existing knowledge and efforts can be made based on the conducted review. In short, we notice that despite significant research and development efforts, runtime patching is still underutilized in practice. For example, an amusingly recent Log4J vulnerability was proposed to be patched by exploiting the vulnerability itself.[17] Furthermore, despite individual software applications using various runtime patching solutions, a lack of general solutions indicates a significant research potential in this domain, specifically developing compiler- and OS kernel-augmented approaches.

We have identified a number of open issues and potential future directions that are shown in Fig. 8 and discussed in the following subsections. Firstly, most of the existing solutions are narrow-scoped and severely limited in their applicability. For instance, a runtime patching solution may employ specific language or execution platform or OS features, restricting the usefulness of such a solution. Alternatively, only specific types of changes (e.g., adding a class method) may be supported by a patching solution, posing challenges for non-trivial modifications.

Secondly, the lack of a systematic approach in runtime patching could be explained by the ad-hoc nature of patch development currently occurring in practice. Unlike traditional software development, which is well established with a multitude of development practices already matured, patch development looks less organized. This is because patches are typically developed under tight time pressure due to the associated security risks. High-profile vulnerabilities can cause high monetary losses, incentivizing the quickest possible patch deployment. This leaves little to no time to approach the task systematically, leading to situations when patches are shortly followed by patches-for-patches (sometimes more than once) to fix the original patches that turned out to be buggy.[18,19]

Lastly, runtime patching capability often comes as an afterthought in contrast to traditional source-level patching. Runtime patches may not even be developed independently but may be automatically generated from developer-supplied source code changes. This leads to a shortage of cooperative patching solutions that could simplify further patching through various degrees of assistance from the existing codebase. Developing a cooperative updating subsystem requires additional efforts at design and development phases. It is thus not commonly being used in practice. Furthermore, foreseeing all possible patching and resource handover scenarios may not be feasible, rendering such implementations insufficiently flexible in the long term.

8.1. Behavior change compatibility

Compatibility between in-flight activities and updated code needs to be maintained to ensure user experience continuity. Patches that introduce significant externally observable changes

10 https://tuxcare.com/live-patching-services/
11 https://ksplice.oracle.com/
12 https://access.redhat.com/solutions/2206511
13 https://github.com/dynup/kpatch
14 https://manpages.ubuntu.com/manpages/trusty/man8/ksplice-create.8.html
15 https://techcommunity.microsoft.com/t5/azure-sql-blog/hot-patching-sql-server-engine-in-azure-sql-database/ba-p/849700
16 https://docs.mulesoft.com/cloudhub-2/ch2-patch-updates
17 https://www.lunasec.io/docs/blog/log4shell-live-patch/
18 https://arstechnica.com/information-technology/2021/12/patch-fixing-critical-log4j-0-day-has-its-own-vulnerability-thats-under-exploit/
19 https://blog.malwarebytes.com/exploits-and-vulnerabilities/2021/08/microsofts-printnightmare-continues-shrugs-off-patch-tuesday-fixes/

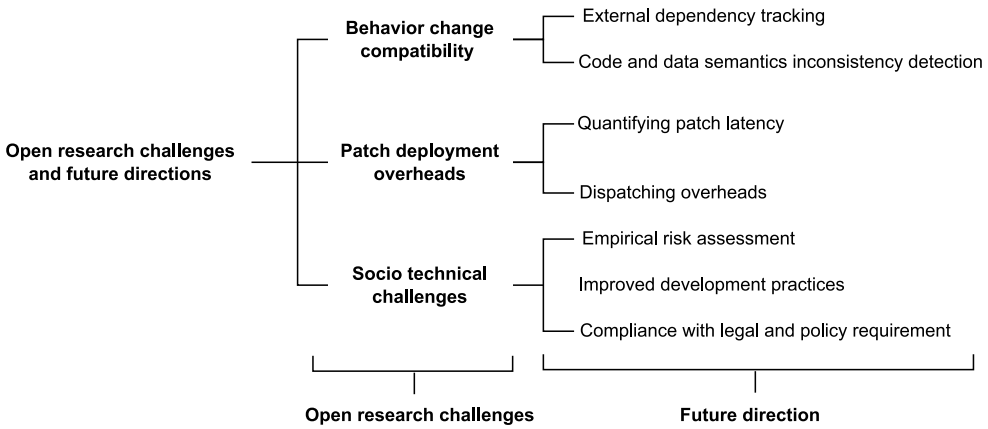


Fig. 8. Open research challenges and potential future directions for runtime patching.

may lead to incompatibilities with the expectation of external observers. Therefore one of the tasks of a run-time patching system is to achieve code and data compatibility or, at the very least, detect potential incompatibilities prior to patch deployment. Dealing with potential incompatibilities between original and patched code requires determining the *scope* (i.e., how far are the patch effects observable) and *type* (i.e., what becomes incompatible) of the incompatibility. For instance, the patch application might be stopped entirely or postponed if certain incompatibilities are detected. Detecting the most disruptive patches can aid in reducing the associated downtime and service interruptions. Considering the mentioned issues of behavior change compatibility, we further discuss the future research directions potentially valuable for solving different aspects of such issues between original and patched code.

8.1.1. External dependencies tracking

Determining where a potential incompatibility lies requires tracking the code's various external dependencies that also need to be updated. For instance, if a function-level patch alters the arguments accepted by the function, all external code fragments that call the function need to be updated accordingly. Moreover, recursive analysis of further code fragments depending on the code that calls patched function is required to propagate the changes as necessary. Similarly, patching a network service application might require updating remote client-side software to retain network-level service compatibility. This process is similar to recursive taint tracking commonly used to track the flow of untrusted user input to identify and rectify unsafe input uses (Ermolinskiy et al., 2010). Generally speaking, security-oriented patches are attempted to be minimized not to change the code behavior drastically. However, it can be argued that with tightly integrated software, even a tiny patch (function- or instruction-level) can lead to drastic side effects. Therefore, understanding where the patch impact would be visible is a crucial preliminary step in runtime patching. Hence, researching and developing quantitative **patch impact estimation** techniques would be helpful in order to detect potential external incompatibilities.

Further patch impact estimation techniques may be used to estimate a given patch's potential disruptiveness. Such complexity metric would enable formally verifying the impact of a given patch. Therefore, simple patches that do not affect end-users can be applied immediately, whereas applying complex and disruptive patches would require planning a downtime. Precisely, the externally-observable software behavior changes would be estimated to determine a given patch impact. Given the increasing trends in software isolation (through containerization and virtualization), special attention needs to be paid to network-related patch side-effects that are observable even on remote network nodes.

8.1.2. Code and data semantics inconsistency detection

When a developer creates the new code fragment, some inconsistencies may be introduced inadvertently. Simple inconsistencies, such as data type mismatch can be detected automatically through type safety preserving approaches. Existing solutions typically attempt to achieve type safety through tracking variable data type changes. In contrast, semantic differences (such as the changed meaning of the variable of the same type) may not be possible to be discovered solely by looking at the changed code fragment. For example, changing an integer variable to a string variable can be detected and rectified accordingly (Bierman et al., 2003).

However, consider a patch that does not alter the variable type (i.e., an integer still stays an integer) but interprets the variable value differently (such as treating meters as feet). Generally, measurement unit handling is prone to such mistakes, and the same distance/angle/weight variable could be interpreted in different ways (e.g., meters vs. feet, degrees vs. radians, pounds vs. kilograms, etc.). Furthermore, a patch that performs a transition from one measurement system to another may not cause a type mismatch error (i.e., violate the type safety) but could still cause issues further down the execution chain. Detecting such semantic changes is unquestionably more complicated as all of the related code paths that rely on the variable outside the scope of the patch must also be analyzed. Unfortunately, no feasible solution aiming to detect semantic inconsistencies has been proposed so far. Perhaps meta-languages similar to μDSU (Cazzola et al., 2018) can be adapted to allow outlining a semantic application layer.

8.2. Patch deployment overheads

In addition to the technical implementation issues, runtime patching overheads must be addressed. These overheads are mainly concentrated in resource (e.g., RAM) and time domains. Coexisting-based solutions imply that multiple copies of code or data must simultaneously be present in memory. Such coexistence might not be an issue for smaller patches; however, applying multiple patches over longer periods may eventually pose challenges. Similarly, resource-constrained IoT devices may also render some solutions impractical due to a lack of hardware resources. For instance, some microchips based on Harvard architecture may not allow direct code modification, leading to the necessity to re-flash main memory contents and restart the execution. Note that, even in such restricted devices, certain extensions have been implemented to aid self-modification that would be useful for patching purposes.[20] Architecture-, device-

[20] http://ww1.microchip.com/downloads/en/AppNotes/doc1644.pdf

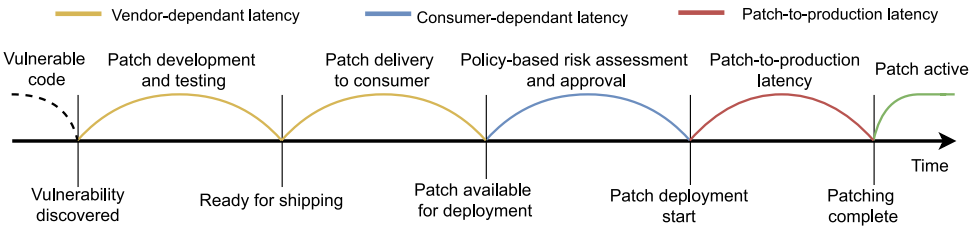


Fig. 9. Patch latency timeline.

and resource-constrained overheads need to be predicted to determine patch suitability for a given system real-time requirement. We identify and discuss the associated quantitative metric research directions below.

8.2.1. Quantifying patch latency

Different patch latency metrics can be defined because of many factors involved in patch preparation, deployment and testing. Fig. 9 shows the different types of patch latency that might be encountered in practice. The *longest time-frame* typically used as a metric of patch refers to the time passed between the discovery of vulnerability or bug and actual update in production. In severe cases, *vendor-to-production latency* can be in the order of magnitude of days to months.[21] This latency, however, does not strictly depend on technical issues and could be dominated by other organizational-level inefficiencies such as supply chain delays, internal company policies, vendor policies and the priority of the system.

Setting aside policy- and development-induced delays, the *patch-to-production* latency metric can be used while considering the technical aspects of runtime patching. This latency essentially covers patch deployment and patch activation steps. However, determining patch-to-production latency numerically is not straightforward, as different strategies cause delays in different aspects. For instance, some solutions may opt to implement co-exist & decay strategy in order to avoid any noticeable system downtime. This, however, means that currently running interactions may not notice the effects of the patch immediately (**activation latency**). Alternatively, state transformation strategies ensure that effects of the patch are observed immediately at the expense of briefly interrupting running interactions (**critical transformation latency**). A more systematic approach in quantifying runtime patching latencies would be beneficial for evaluation and direct comparison of different patching implementation approaches.

8.2.2. Dispatching overheads

Code path or resource dispatching approaches attempt to minimize software system overhead. However, even lightweight dispatching overheads tend accumulate over multiple repeated patches causing increased latencies. While the *decay* phase is designed to release an unused resource, both additional memory and time requirements may grow for long-living services. In other words, if external entities keep using the resource for a prolonged period of time, the resource would not be released, leading to a higher amount of resources such as memory, network sockets or file descriptors being used. While focusing on immediate gains, some existing solutions tend to overlook some long-living software usage scenarios during their evaluation to estimate the effects of accumulated overheads (Chen et al., 2007). Investigating dispatching overheads in relation to software system usage under high load would provide a better understanding of patching implementation applicability in practice.

[21] https://www.businessinsider.com.au/zoom-security-flaw-hackathon-dropbox-2020-4?r=US&IR=T

8.3. Socio-technical challenges

In addition to purely technical issues related to runtime patching, several socio-technical aspects may pose additional complications. For instance, delays due to imposing internal policies or legislation may significantly slow down the patching procedures. Organizations may consider improving software development practices that incorporate runtime patching at the design phase. However, the monetary and time costs associated with rewriting large legacy software packages according to improved development practices may drive off software developers. Furthermore, in some cases, extra complications in patching processes may be induced by supply chain participants (such as mobile service providers or hardware vendors) for the purposes of patch verification or secure distribution. We identify some potentially beneficial research directions to solve socio-technical challenges hindering wider adoption of runtime patching.

8.3.1. Empirical risk assessment

Empirical evaluation of the potential risks of deploying runtime patches requires significant attention from a business perspective. Organizations need to predict potential patch consequences regarding service disruption, data incompatibility, impact on adjacent services or even new vulnerabilities introduced by a patch. While ultimately, business analysts may make the final decision manually, some automated quantitative metric determination algorithms can simplify their work. For instance, automatically deducing the list of third-party software services that might be affected by a patch or predicting the service disruption length would greatly aid in making an informed decision backed by actual data. Therefore, researching and developing quantitative patch impact prediction and estimation algorithms at a business scale would be helpful in practice. In addition, another potentially beneficial research direction is the optimization of concrete risk assessment procedures employed in a given company to minimize lengthy delays caused due to following potentially complicated internal policies (Dissanayake et al., 2022).

8.3.2. Improved development practices

A group of socio-technical challenges related to runtime patching arises from economic factors such as monetary and time constraints. Notably, some existing solutions implement improved development practices, programming languages or supporting frameworks that significantly simplify runtime patching at the technical level. However, these solutions did not attract highly due to the hidden (non-technical) costs. Firstly, using these solutions requires learning new tools. Secondly, achieving compatibility between existing legacy code and the patching framework requirements typically requires a certain level of code modification. For instance, identifying and defining safe update points within existing code for a developer may be complicated in high-load applications. Combining these two reasons typically lead to high costs of employing such solutions in practice. Thus, a comprehensive study aimed at software development practitioners

in different fields might provide additional clues to increasing the adoption of design-time runtime patching solutions. Another pragmatic aspect to explore is the potential avenues in popularizing such design-time patching approaches among a wider audience, specifically targeting software developers.

8.3.3. Compliance with legal and policy requirements

Lastly, the least technical group of issues complicating runtime patching revolve around external factors like local legislation and external business partners' interaction. Patch delivery and distribution channels may impose organizational complications and significant delays. For instance, patching Android OS needs to overcome organizational vendor-imposed limitations such as obtaining digital signatures from all parties involved in the supply chain (Chen et al., 2018). In addition, highly regulated organizations such as health-, finance- or defense-related may have additional restrictions imposed by specific legislation and regulations,[22] requiring to conduct patching strictly according to predefined procedures. While being out of the scope of this study, legislation and regulation improvement may also benefit from the body of knowledge generated by the patch impact prediction research line.

9. Conclusion

Runtime patching techniques and approaches require radical transformation following the emerging technologies in safety and mission-critical system adoption. Deploying and activating patches with minimum overhead and downtime are crucial steps of applying patches at runtime. We present a taxonomy highlighting the four key aspects that need consideration for runtime patch deployment. We identify and analyze the seven patch granularity levels along with two general patching strategies (state transformation and co-exists & decay) and three responsible entities (vendor, consumer and third parties). We further define a high-level workflow (Fig. 5) of applying and activating patches supporting the four identified key implementation capabilities that are (i) memory management, (ii) resource transformation, (iii) safe execution state determination and (iv) execution path dispatching. Finally, the review highlights the challenges and suggests potential future prospects. We envision the area mainly needs a systematic approach to develop practical and convenient patching solutions applicable in a wide variety of infrastructures, programming languages and execution environments. The area further requires defining quantifiable metrics to identify the impact of a patch by semantics and dependency tracking solutions. We observe many open opportunities and concerns that need to be considered by vendors, end-users and third parties.

Future research lines include obtaining patching-related real-world data through a practical field study. Analyzing this data would enable understanding the statistics of existing patching challenges in various critical domains. Notable research questions should focus on the patching procedures commonly applied in practice, patch granularity distribution, the complexity of the patches applied, patch-induced system downtime and service continuity, as well as potential incompatibility issues. To conclude, an in-depth understanding of the issues and concerns of practitioner and business requirements is crucial to bridge the gap between runtime patching research and practical implementation/deployment in production.

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.

[22] https://techcrunch.com/2022/01/05/ftc-legal-action-log4j/

Data availability

No data was used for the research described in the article.

Acknowledgments

The work has been partially supported by the Cyber Security Cooperative Research Centre Limited whose activities are partially funded by the Australian Government's Cooperative Research Centres Programme.

References

Ahmed, B.H., Lee, S.P., Su, M.T., Zakari, A., 2020. Dynamic software updating: a systematic mapping study. IET Softw. 14 (5), 468–481.

Araujo, F., Taylor, T., 2020. Improving cybersecurity hygiene through JIT patching. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA, pp. 1421–1432. http://dx.doi.org/10.1145/3368089.3417056.

Arnold, J., Kaashoek, M.F., 2009. Ksplice: Automatic rebootless kernel updates. In: Proceedings of the 4th ACM European Conference on Computer Systems. pp. 187–198.

Ayres, N., Deka, L., Paluszczyszyn, D., 2021. Continuous automotive software updates through container image layers. Electronics 10 (6), 739.

Bierman, G., Hicks, M., Sewell, P., Stoyle, G., 2003. Formalizing dynamic software updating. In: Proceedings of the Second International Workshop on Unanticipated Software Evolution. USE, Citeseer, pp. 1–17.

Cazzola, W., Chitchyan, R., Rashid, A., Shaqiri, A., 2018. μ-DSU: a micro-language based approach to dynamic software updating. Comput. Lang. Syst. Struct. 51, 71–89.

Chamith, B., Svensson, B.J., Dalessandro, L., Newton, R.R., 2016. Living on the edge: Rapid-toggling probes with cross-modification on x86. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 16–26.

Chen, G., Jin, H., Zou, D., Liang, Z., Zhou, B.B., Wang, H., 2015. A framework for practical dynamic software updating. IEEE Trans. Parallel Distrib. Syst. 27 (4), 941–950.

Chen, G., Jin, H., Zou, D., Zhou, B.B., Liang, Z., Zheng, W., Shi, X., 2013. Safestack: Automatically patching stack-based buffer overflow vulnerabilities. IEEE Trans. Dependable Secure Comput. 10 (6), 368–379.

Chen, Y., Li, Y., Lu, L., Lin, Y.-H., Vijayakumar, H., Wang, Z., Ou, X., 2018. Instaguard: Instantly deployable hot-patches for vulnerable system programs on android. In: 2018 Network and Distributed System Security Symposium (NDSS'18).

Chen, H., Yu, J., Chen, R., Zang, B., Yew, P.-C., 2007. Polus: A powerful live updating system. In: 29th International Conference on Software Engineering (ICSE'07). IEEE, pp. 271–281.

Chen, Y., Zhang, Y., Wang, Z., Xia, L., Bao, C., Wei, T., 2017. Adaptive android kernel live patching. In: 26th {USENIX} Security Symposium ({USENIX} Security 17). pp. 1253–1270.

Dissanayake, N., Jayatilaka, A., Zahedi, M., Babar, M.A., 2022. Software security patch management - a systematic literature review of challenges, approaches, tools and practices. Inf. Softw. Technol. 144, 106771. http://dx.doi.org/10.1016/j.infsof.2021.106771, URL https://www.sciencedirect.com/science/article/pii/S0950584921002147.

Doddamani, S., Sinha, P., Lu, H., Cheng, T.-H.K., Bagdi, H.H., Gopalan, K., 2019. Fast and live hypervisor replacement. In: Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. In: VEE 2019, Association for Computing Machinery, New York, NY, USA, pp. 45–58. http://dx.doi.org/10.1145/3313808.3313821.

Duan, R., Bijlani, A., Ji, Y., Alrawi, O., Xiong, Y., Ike, M., Saltaformaggio, B., Lee, W., 2019. Automating patching of vulnerable open-source software versions in application binaries. In: NDSS.

Emelyanov, P., Kolyshkin, K., 2011. Checkpoint/restart (mostly) in user space. In: 2011 Linux Plumbers Conference.

Ermolinskiy, A., Katti, S., Shenker, S., Fowler, L., McCauley, M., 2010. Towards practical taint tracking. Tech. Rep., Citeseer.

Gartner, 2019. Ensure Cost Balances With Risk in High-Availability Data Centers.

Giuffrida, C., Kuijsten, A., Tanenbaum, A.S., 2013. Safe and automatic live update for operating systems. ACM Sigplan Notices 48 (4), 279–292.

Gregersen, A.R., Rasmussen, M., Jørgensen, B.N.r., 2013. State of the art of dynamic software updating in java. In: International Conference on Software Technologies. Springer, pp. 99–113.

Gu, Y., Lin, Z., 2016. Derandomizing kernel address space layout for memory introspection and forensics. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. pp. 62–72.

Gupta, D., Jalote, P., Barua, G., 1996. A formal framework for on-line software version change. IEEE Trans. Softw. Eng. 22 (2), 120–131.
Hayden, C.M., Saur, K., Smith, E.K., Hicks, M., Foster, J.S., 2014. Kitsune: Efficient, general-purpose dynamic software updating for C. ACM Trans. Program. Lang. Syst. 36 (4), 1–38.
Hayden, C.M., Smith, E.K., Denchev, M., Hicks, M., Foster, J.S., 2012. Kitsune: Efficient, general-purpose dynamic software updating for C. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 249–264.
Hicks, M., Nettles, S., 2005. Dynamic software updating. ACM Trans. Program. Lang. Syst. 27 (6), 1049–1096. http://dx.doi.org/10.1145/1108970.1108971.
Hosek, P., Cadar, C., 2015. Varan the unbelievable: An efficient n-version execution framework. ACM SIGARCH Comput. Archit. News 43 (1), 339–353.
Hu, Y., Zhang, Y., Gu, D., 2019. Automatically patching vulnerabilities of binary programs via code transfer from correct versions. IEEE Access 7, 28170–28184.
Ilvonen, V., Ihantola, P., Mikkonen, T., 2016. Dynamic software updating techniques in practice and educator's guides: a review. In: 2016 IEEE 29th International Conference on Software Engineering Education and Training. CSEET, IEEE, pp. 86–90.
Jeong, H., Baik, J., Kang, K., 2017. Functional level hot-patching platform for executable and linkable format binaries. In: 2017 IEEE International Conference on Systems, Man, and Cybernetics. SMC, IEEE, pp. 489–494.
Kashyap, S., Min, C., Lee, B., Kim, T., Emelyanov, P., 2016. Instant {OS} updates via userspace checkpoint-and-restart. In: 2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16). pp. 605–619.
Le, D., Xiao, J., Huang, H., Wang, H., 2014. Shadow patching: Minimizing maintenance windows in a virtualized enterprise environment. In: 10th International Conference on Network and Service Management (CNSM) and Workshop. IEEE, pp. 169–174.
Lee, S.-H., Kim, S.-H., Hwang, J.Y., Kim, S., Jin, S.-H., 2020. Is your android app insecure? Patching security functions with dynamic policy based on a java reflection technique. IEEE Access 8, 83248–83264.
Lopez, J., Babun, L., Aksu, H., Uluagac, A.S., 2017. A survey on function and system call hooking approaches. J. Hardw. Syst. Secur. 1 (2), 114–136.
Lucia, B., Balaji, V., Colin, A., Maeng, K., Ruppel, E., 2017. Intermittent computing: Challenges and opportunities. In: 2nd Summit on Advances in Programming Languages (SNAPL 2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
Miedes, E., Muñoz-Escoí, F., 2012. A survey about dynamic software updating. In: Instituto Universitario Mixto Tecnologico de Informatica, Universitat Politecnica de Valencia, Campus de Vera S/N. 46022.
Mugarza, I., Parra, J., Jacob, E., 2018. Analysis of existing dynamic software updating techniques for safe and secure industrial control systems. Int. J. Safety Secur. Eng. 8 (1), 121–131.
Mugarza, I., Parra, J., Jacob, E., 2020. Cetratus: A framework for zero downtime secure software updates in safety-critical systems. Softw. - Pract. Exp. 50 (8), 1399–1424.
Neamtiu, I., Hicks, M., Stoyle, G., Oriol, M., 2006. Practical dynamic software updating for C. ACM SIGPLAN Notices 41 (6), 72–83.
Orso, A., Rao, A., Harrold, M., 2002. A technique for dynamic updating of java software. In: International Conference on Software Maintenance, 2002. Proceedings. pp. 649–658. http://dx.doi.org/10.1109/ICSM.2002.1167829.
Payer, M., Bluntschli, B., Gross, T.R., 2013. DynSec: On-the-fly code rewriting and repair. In: 5th Workshop on Hot Topics in Software Upgrades (HotSWUp 13).
Payer, M., Gross, T.R., 2013. Hot-patching a web server: A case study of asap code repair. In: 2013 Eleventh Annual Conference on Privacy, Security and Trust. IEEE, pp. 143–150.
Pina, L., Andronidis, A., Hicks, M., Cadar, C., 2019. Mvedsua: Higher availability dynamic software updates via multi-version execution. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 573–585.
Pina, L., Hicks, M., 2016a. Tedsuto: A general framework for testing dynamic software updates. In: 2016 IEEE International Conference on Software Testing, Verification and Validation. ICST, IEEE, pp. 278–287.
Pina, L., Hicks, M., 2016b. Tedsuto: A general framework for testing dynamic software updates. In: 2016 IEEE International Conference on Software Testing, Verification and Validation. ICST, IEEE, pp. 278–287.
Ramaswamy, A., Bratus, S., Smith, S.W., Locasto, M.E., 2010. Katana: A hot patching framework for elf executables. In: 2010 International Conference on Availability, Reliability and Security. IEEE, pp. 507–512.
Rommel, F., Dietrich, C., Rodin, M., Lohmann, D., 2019a. Multiverse: Compiler-assisted management of dynamic variability in low-level system software. In: Proceedings of the Fourteenth EuroSys Conference 2019. pp. 1–13.
Rommel, F., Glauer, L., Dietrich, C., Lohmann, D., 2019b. Wait-free code patching of multi-threaded processes. In: Proceedings of the 10th Workshop on Programming Languages and Operating Systems. pp. 23–29.
Rothberg, V., Dietrich, C., Graf, A., Lohmann, D., 2016. Function multiverses for dynamic variability. In: 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W). IEEE, pp. 1–5.
Ruckebusch, P., De Poorter, E., Fortuna, C., Moerman, I., 2016a. Gitar: Generic extension for internet-of-things architectures enabling dynamic updates of network and application modules. Ad Hoc Netw. 36, 127–151.
Ruckebusch, P., De Poorter, E., Fortuna, C., Moerman, I., 2016b. Gitar: Generic extension for internet-of-things architectures enabling dynamic updates of network and application modules. Ad Hoc Netw. 36, 127–151.
Salls, C., Shoshitaishvili, Y., Stephens, N., Kruegel, C., Vigna, G., 2017. Piston: Uncooperative remote runtime patching. In: Proceedings of the 33rd Annual Computer Security Applications Conference. pp. 141–153.
Seifzadeh, H., Abolhassani, H., Moshkenani, M.S., 2013. A survey of dynamic sofstware updating. J. Softw.: Evol. Process 25 (5), 535–568.
Siniavine, M., Goel, A., 2013. Seamless kernel updates. In: 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks. DSN, IEEE, pp. 1–12.
Sun, J., Wu, C., Ye, J., 2020. Blockchain-based automated container cloud security enhancement system. In: 2020 IEEE International Conference on Smart Cloud (SmartCloud). IEEE, pp. 1–6.
Suse, 2021. Live patching the linux kernel using kgraft. URL https://documentation.suse.com/sles/12-SP4/html/SLES-kgraft/index.html, Last accessed 30 November 2021.
Tunde-Onadele, O., Lin, Y., He, J., Gu, X., 2020a. Self-patch: Beyond patch tuesday for containerized applications. In: 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems. ACSOS, IEEE, pp. 21–27.
Tunde-Onadele, O., Lin, Y., He, J., Gu, X., 2020b. Toward just-in-time patching for containerized applications. In: Proceedings of the 7th Symposium on Hot Topics in the Science of Security. pp. 1–2.
Xu, Z., Zhang, Y., Zheng, L., Xia, L., Bao, C., Wang, Z., Liu, Y., 2020. Automatic hot patch generation for android kernels. In: 29th {USENIX} Security Symposium ({USENIX} Security 20). pp. 2397–2414.
Zhang, H., Zhao, L., Xu, L., Wang, L., Wu, D., 2014. Vpatcher: Vmi-based transparent data patching to secure software in the cloud. In: 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications. IEEE, pp. 943–948.
Zhang, X., Zheng, X., Wang, Z., Li, Q., Fu, J., Zhang, Y., Shen, Y., 2019. Fast and scalable VMM live upgrade in large cloud infrastructure. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 93–105.
Zhao, Z., Gu, T., Ma, X., Xu, C., Lü, J., 2016. Cure: Automated patch generation for dynamic software update. In: 2016 23rd Asia-Pacific Software Engineering Conference. APSEC, IEEE, pp. 249–256.
Zhou, L., Zhang, F., Liao, J., Ning, Z., Xiao, J., Leach, K., Weimer, W., Wang, G., 2020. Kshot: Live kernel patching with SMM and SGX. In: 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. DSN, IEEE, pp. 1–13.

Chadni Islam is a postdoctoral researcher in CREST-Centre for Research on Engineering Software Technologies, School of Computer Science, University of Adelaide. She completed her Ph.D. degree in 2020 under the supervision of Professor M. Ali Babar from the same school. She was co-supervised by Dr. Surya Nepal from CSIRO's Data61. Her research expertise falls at the intersection of Cyber Security and Software Engineering. Her Ph.D. research was focused on providing architecture support for security orchestration and automation systems using advanced software engineering technologies. She is interested in leveraging existing software engineering, analytical reasoning, natural language processing and machine learning tools and techniques to develop an intelligent self-adaptive security orchestration and automation platform.

Victor Prokhorenko is a researcher with the Centre for Research on Engineering Software Technologies (CREST) at the University of Adelaide. Victor has more than 14 years of experience in software engineering with primary areas of expertise, including investigation of technologies related to software resilience, trust management and big data solutions hosted within the OpenStack private cloud platform. Victor has obtained a Ph.D. in Computer Science from the University of South Australia.

M. Ali Babar is a Professor in the School of Computer Science, University of Adelaide. He is an honorary visiting professor at the Software Institute, Nanjing University, China. He has authored/co-authored more than 230 peer-reviewed papers in premier Software Technology journals and conferences. With an H-Index 48, the level of citations to his publications is among the leading Software Engineering researchers in Aus/NZ. At the University of Adelaide, Professor Babar has established an interdisciplinary research centre, CREST-Centre for Research on Engineering Software Technologies, where he leads the research and research training of more than 30 members. He has been involved in attracting several millions of dollar worth of research resources over the last ten years. Prof Babar leads the University of Adelaide's participation in the Cyber Security Cooperative Research Centre (CSCRC), one of the largest Cyber Security initiative of the Australasian region. Within the CSCRC, he leads the theme on Platform and Architecture for Cyber Security as a Service. Further details can be found on: https://researchers.adelaide.edu.au/profile/ali.babar#my-research.