Hot Patching Hot Fixes: Reflection and Perspectives

Carol Hanna
Department of Computer Science
University College London
London, United Kingdom
carol.hanna.21@ucl.ac.uk

Justyna Petke
Department of Computer Science
University College London
London, United Kingdom
j.petke@ucl.ac.uk

***Abstract*—With our reliance on software continuously increasing, it is of utmost importance that it be reliable. However, complete prevention of bugs in live systems is unfortunately an impossible task due to time constraints, incomplete testing, and developers not having knowledge of the full stack. As a result, mitigating risks for systems in production through hot patching and hot fixing has become an integral part of software development. In this paper, we first give an overview of the terminology used in the literature for research on this topic. Subsequently, we build upon these findings and present our vision for an automated framework for predicting and mitigating critical software issues at runtime. Our framework combines hot patching and hot fixing research from multiple fields, in particular: software defect and vulnerability prediction, automated test generation and repair, as well as runtime patching. We hope that our vision inspires research collaboration between the different communities.**

***Index Terms*—Software Engineering, Software maintenance, Predictive maintenance, Prediction methods, Repair**

I. Introduction

Ensuring that deployed software is completely bug-free is almost always infeasible. Program verification is an undecidable problem [1], and deployment of software verification techniques comes with many challenges [2], [3]. In practice, software bugs are thus typically discovered through testing [4]. However, as quoted by Dijkstra, testing is incomplete and can only show the presence not the absence of bugs [5]. Moreover, even when bugs are discovered prior to deployment, developers do not always have time to fix them due to tight release deadlines. Post-production bugs are very costly to enterprises [6] as they affect system availability and robustness. In user-facing systems, this may damage the enterprise's reputation. Therefore, when discovered, these types of bugs are of the highest importance to all stakeholders.

To understand the space of research that tackles such critical bugs, we started with a literature survey. We used two keywords commonly used to refer to the software engineering activity of patching critical, post-production bugs: "hot fix" and "hot patch". In doing so, we quickly noticed that the definitions in existing work are inconsistent which has diverged the research on the topic into two distinctly different fields. The first focuses on the time-criticality aspect of patching the bug in the definition, leading the work down the path of how this patching process can be more efficient [7]–[9]. The second category delves into helping developers integrate changes dynamically, at runtime, without causing downtime to the deployed system [10]–[12]. This second set of papers focuses more on binary-level patching and its dynamic propagation into live systems. This split between the definitions creates a separation between the research that aligns with each. We thus provide a taxonomy of all of the definitions found and reflect on how associated research could drive future work.

Our survey of literature on hot patching and hot fixing revealed research in different silos. Existing works focus on either the detection of software defects and vulnerabilities, their mitigation, or the propagation of the patches onto a live system. Moreover, they only target a specific type of software issue (e.g., security vulnerability) or end system (e.g., kernel vs. application specific). We pose that research on hot patching and hot fixing can be represented in a unified framework, with an ultimate goal of creating an end-to-end automated solution that fully automates the process of predicting and mitigating issues in post-production software.

To realise our vision, we draw inspiration from several fields which until now have been somewhat orthogonal such as software testing [13], defect and vulnerability prediction [14] [15], automated program repair [16], binary analysis [17], data mining [18], and runtime patching [19]. Combing predictive modeling with automated program repair specifically has started to receive attention in the last few years with Nowack et al. [20] proposing tooling for predicting and fixing bugs within Java programs. Most recently Harman [21] posed that existing techniques should be explored for the "hot fix problem", i.e, the need to discover and deploy fixes in real time. We aim to address this problem.

Drawing upon results of our survey on hot patching and hot fixing, we propose a fully automated and extensible framework (Figure 1) that predicts *critical* software issues, provides mitigation solutions in the form of a patch, which is then applied to the software system *at runtime*. We outline how each of the components can be realised.

By clarifiying terminology on hot patching and hot fixing, and presenting a unified framework, we hope to inspire new

This work was supported by EPSRC grant EP/P023991/1. For the purpose of open access, the author(s) has applied a Creative Commons Attribution (CC BY) license to any Accepted Manuscript version arising.

research directions and idea pollination between the different communities. The challenges will come from fitting the components from the different research fields together and do so in a scalable way. Our ultimate goal is realisation of the presented framework to increase reliability of production software.

II. Survey Methodology

To gather information on patching critical issues, that typically bypass the planned software release schedule, we conducted a literature survey using a primary search on two keywords: "hot patch" and "hot fix". The search was conducted over four popular computer science search engines: IEEE Xplore, ACM Digital Library, ScienceDirect, and the DBLP Computer Science Bibiliography. We found 547 search results for "hot fix" and 475 results for "hot patch", with 136 relevant to patching software. The scope that we used for relevancy encapsulated all conferences, workshops, and journal papers as well as PhD theses published by 21/2/2023. Overall, we found 21 definitions for "hot fix" and 14 definitions for "hot patch"in our searches, in 33 distinct papers. We present these in the next section, pointing out differences and commonalities.

III. Hot Patch or Hot Fix? Terminology Evolution

Our survey revealed that the origin of the terms "hot patch" and "hot fix" is unclear. It seems that originally the term *hot* in "hot patch" was intended to mean the liveness of the system in which the patch was being deployed. Given this, hot patching would mean patch deployment into running systems. However, this more literal definition began to shift in different directions that reflect the different stakeholders' perspectives regarding hot patching activities. More specifically, for software developers, when a code change needs to be integrated into a live system, it must be a critical time-sensitive change. This is because otherwise it would simply be developed within the planned software release cycle. This is what we hypothesise created a split into two different definitions, described below, which we see in the literature today.

The first definition is what you may traditionally think of when a hot patch is mentioned in a conversation: a patch for a time-critical bug. This patch is usually small and temporary and needs to be deployed quickly to mitigate specific critical bugs. The second category addresses the dynamic property needed to integrate patches during program's runtime. Research in the second category focuses on binary modifications to live programs, most commonly using techniques such as binary quilting [22]. The majority of research in this area is in the security and vulnerability mitigation domain. Finally, we found some field-specific definitions which place hot patching within a very specific context.

Our review revealed that the majority of "hot fix" terminology addresses the first category of definitions which targets the criticality element of this patching process. Whereas "hot patch" is a term more widely used to address the dynamic quality. However, there were some inconsistencies with this as well, i.e., Popovici et al. [23] refer to a *hot-fix* as "an extension applied to a running application server to modify the behavior of a large number of running components" and Limoncelli [24] refers to *hot patches* as those that target deployed high-priority bugs. Thus, we will be using the term *hot fix* in Section III-A. *hot patch* in Section III-B, and the term syntax as it appears in the specific paper for Section III-C.

A. Criticality Definition

Definitions that address hot fixes solely as those that remedy pressing bugs that need immediate attention capture four different properties: patch size, patch permanency, bug criticality, and production status of the target system. The consensus across all definitions that address the patch size property is that a hot fix must be small in size as opposed to a service pack or a full new software version release [25] [26] [27] [28] [29] [30]. Casco [31] specifies that a hot fix must address one specific issue. Gupta et al. [32] mentions that it must be limited to either one or two. Agarwal and Garg [33] and Stanger [34]'s definitions leave this more ambiguous, stating that a hot fix targets multiple issues. Hot fixes are intended to be deployed temporarily while a permanent patch is being developed [35]. This is due to the time criticality element which limits the patch from being extensively tested before its integration into the system. Moreover, the intent of the patch is generally to mitigate the symptom of the critical bug rather than fully resolving its root cause as that would be more time-consuming and might even require some system redesign. While the criticality of the bug that the hot fix addresses is a property that is implied in this category of definitions, it's not always explicitly stated. While some definitions mention that hot fixes are deployed to systems in production [24] [36], this property is not always directly conveyed. Karale et al. [37] note that hot fixes don't need to always be released publicly. Finally, Bailey [38] regards hot fixes as Microsoft patches not included in the standard service pack.

To sum up, the majority of the literature in this category regards a hot fix as a small, temporary improvement to a specific critical issue in a deployed system. The following definition from the software security domain captures this well:

> *"A hotfix is a small patch file, generally targeted to one or two specific issues. Hotfixes are usually developed and released in a short timeframe, with less testing than is done for service packs"(Gupta et al. [32])*

B. Run-time Definition

The second major category of definitions found regards hot patches as the patching of software dynamically [39] [10] [11] [40] [41] [42] [43]. This is especially prevalent for systems that have high availability requirements and require bug patches to be injected during runtime without having to reboot the system. An interesting contradiction that we found here is regarding the size of the patch. As previously explained, the first category of definitions highlights that hot patches must be small in size and specifically target a limited number of bugs. However, in the context of the dynamic

definition, Popovici et al. [23] define a hot patch as an upgrade that modifies a large number of components in a server during runtime. Russinovich et al. [12] detail limitations of this technique, e.g., that it only supports function-level updates.

We found the following definition of a hot patch to best encapsulate the most common usage of the term in the literature, as described above:

> *"Hot-patching consists of applying a patch to running software without requiring that the software or system be restarted." (Reffett and Fleck [40])*

C. Domain-Specific Definition

Additionally, depending on the domain of the research, some work provides field-specific definitions. Han et al. [44] define *hot-fix* from the perspective of an RNN-based speech recognition system where they expect a query's results to be only minimally affected by a hot patch. Hargrave's Communications dictionary [45] describes hot fixes as migrating disk sectors within the LAN. In the domain of operating systems specifically, hot patching is defined as deploying patches within the kernel dynamically [46]. In the context of software testing, Chen et al. [47] regard a hotfix as an implementation in which a code section is wrapped so that during execution the program can be updated to the buggy version. Hotfixes have also been defined in the domain of information retrieval. Here, the top items are shuffled and then ranked based on feedback from user clicks [48]. For firmware, hot-patching is the ability to patch firmware to a device while maintaining the availability of the system [49]. Finally, Illes-Seifert et al. [50] look at hotfixes from a different lens where they consider an update to be a hotfix if it was integrated within the first 5% of time between two consecutive version releases.

Chen et al. [51], provide a definition that combines both of the aforementioned definition categories of time-sensitivity and during runtime integration. Although their paper focuses on security vulnerability patching and their definition is bound within that scope, it does capture and summarize all of the properties found in previous definitions.

IV. Vision for an End-to-End Automated Solution

Our survey of the literature revealed that the terms "hot patch" and "hot fix" have been used in different contexts in different domains. This conclusion incited us to propose a unified framework that we hope will foster collaboration and the flow of ideas between the different research fields.

Our vision entails a fully-automated end-to-end framework for the treatment of software vulnerabilities and defects (which we refer to as *risks* here for brevity) in post-production systems. We present it in Figure 1. The framework is composed of three major parts: **detection** of software risks that need to be mitigated, **mitigation** techniques to generate patches for those risks, and finally **propagation** of the patches into the system during runtime.

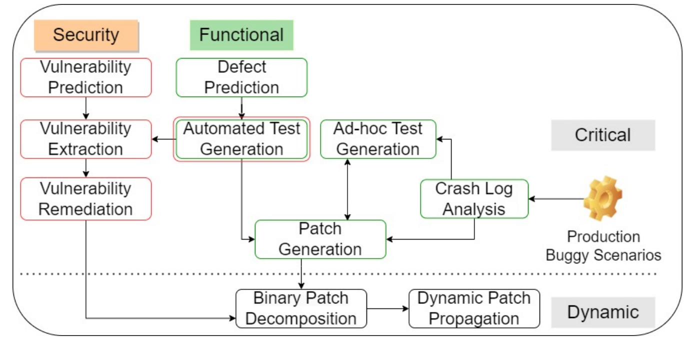

Fig. 1. End-to-end Framework for Software Risk Detection & Mitigation

In the following sections, we will present each of the modules and detail their function within the framework. We expand on relevant work that could be utilised in realizing each of the components, making the implementation of the framework given the currently available techniques in the literature feasible. The framework is extensible and allows for the addition of new modules as well as for modules to combine multiple techniques to achieve their intended goal.

A. Defect Prediction

When a critical bug is discovered through a failure in the live system or through reports by users, developers have a very limited time to generate a patch. The longer it takes for a patch to get deployed, the more monetary and reputational damage gets caused to the enterprise. The goal of our framework is to minimise that window as much as possible. One way of doing so is integrating a defect prediction module that scans the system and foresees likely defective code snippets. Multiple techniques already exist for defect prediction [14]. These predicted code snippets can then become the target for more extensive testing. If the testing then reveals the defect, patch generation efforts can be directed towards it to hot patch the system before the defect manifests in production.

B. Automated Test Generation

The purpose of this module is to use to generate automated tests for the code sections that the defect prediction module foresees as defective. Once a code section is predicted to be defective, we generate tests that reveal this defect, to aid patch generation strategies (Section IV-H). We will use defect prediction analysis to build an oracle for test generation. Several automated test generation techniques already exist which can be used in combination with each other to increase the likelihood of revealing the defect [52]. Additionally, automated hypertest generation [53] to detect security vulnerabilities can be utilised here. Mesecan et al. [53] focus on information leakage, though the techniques could be extended to detect side-channel attacks, for instance.

C. Vulnerability Prediction

To be able to hot patch security weaknesses, software developers must get ahead of the attackers in detecting the system's vulnerabilities. This will allow the developers to have

the time to generate the patch and deploy it. Thus, we see a vulnerability prediction component as a part of our vision for an end-to-end framework for hot patching live systems. There is already abundant research on security vulnerability prediction which can be utilised in making up this component [15].

Tunde-Onadele et al. [54] present a concrete example of how vulnerability prediction can be integrated as tooling for the context of hot patching security holes. Their work presents an anomaly detection module that combines system call tracing and an unsupervised machine learning algorithm to detect vulnerability exploitation.

D. Vulnerability Extraction

Given the outputs of the vulnerability prediction module, we need a component that correlates the predicted exploitation with known vulnerabilities. This will allow us to automate the patching of the security hole by applying the known vulnerability patch to the predicted exploit. In continuation of the work by Tunde-Onadele et al. [54] (Section IV-C), they extract the most frequently appeared system calls during the exploit period to identify the specific vulnerability. Mesecan et al. [53] present a dynamic algorithm for information leak localization guided by hypertests (to be generated in the automated test generation module). Using this, information leakage vulnerabilities can be extracted, to be mitigated in the vulnerability remediation module.

E. Vulnerability Remediation

If the vulnerability extraction module was able to match the exploit with a known vulnerability, then the remediation of the vulnerability becomes easy to automate. We simply need a module to automate the application of the released update that patches this vulnerability. For instance, Bard et al. [55] present a framework that can automatically insert vulnerability mitigations.

F. Ad-hoc Test Generation

The main purpose of this component is to ease the reproduction of the bug in the development environment. This module is inspired by Saieva et al. [56]. In their work, they replay the inputs from the production environment while accounting for non-determinism. Once a patch is generated, this component can behave as a second layer of verification for checking whether the generated patch does indeed mitigate the symptom as it was observed in the production environment.

G. Log Analysis

The log analysis module processes the crash logs from the production environment (most commonly from the users themselves, but in non-user-facing systems this is not always true). Crash logs are hard to decipher manually, so a component like this will help in two ways. Firstly, it will parse the log, which will make it easier for the software developers to make sense of the crash and thus meet the criticality requirement for generating a hot patch manually if needed more quickly. Secondly, it will aid in the automation of producing the patch as its output can inform the patch generation module. Existing work can inspire this component, such as "The Crash Analyzer" [7] which aggregates crash logs, mines crash patterns, and generates reproducible scenarios.

H. Patch Generation

For optimised efficacy and effectiveness, the patch generation component would combine multiple automated program repair techniques. Current approaches in automated program repair can be categorised into three types: constraint-based, learning-based, and heuristic-based techniques [57]. The combination of these repair strategies would complement each other as each strategy has its strengths and weaknesses. The test-based techniques will be able to successfully integrate here due to the automated test generation component that we include in the framework (recall Section IV-B).

As per the definition of a hot patch, the generated patches are meant to be temporary. The purpose of a hot patch is to mitigate the symptoms of the defect temporarily while a more permanent patch is being developed. The reason for this is the time criticality element which prevents developers from having the time to fix the root cause of the issue or the time to extensively test the patch.

An example that would work well in this module to temporarily mitigate failure symptoms is genetic-based program repair [58] [59] [60]. With increasing abilities of large-language models [61], these can be used for quick patch generation, which will be verified with testing. Another idea that can be adopted here is the work by Gomez et al. [7]. They describe a high-level patch generator similarly but specifically present the "Patcher" module which takes system traces (outputs of a module similar to the log analysis component from Section IV-G) and wraps try-catch blocks around suspicious methods as a temporary fix.

I. Dynamic Patch Propagation

Once an acceptable patch is generated, we must be able to deploy it into the system dynamically. The way to propagate the patch will depend on the type of system that we are deploying into. Various techniques for different types of systems exist [19], e.g. Gómez et al. [7] use Dexpler [62] and the Vulmetis [63] tooling which converts patches into hot patches using weakest precondition reasoning.

J. Binary Patch Decomposition

One of the major risks of hot patching specifically and software upgrades generally is breaking functionality due to incompatibility [56]. Thus, the purpose of this module is to alleviate this friction. The binary decomposition algorithm [56] makes this possible by breaking down the patch into its building blocks. Developers will include metadata with the patch that details the dependencies of the patch between versions, allowing partial deployment of the patch into the live system if necessary.

V. Conclusions

In this paper, we discuss the crucial task in the software engineering life cycle of alleviating the symptoms of software problems in live systems. We find that there are contradictions in terminology in the literature on this topic. Thus, we begin by providing a taxonomy of the coined terms for this task, *hot patch* and *hot fix*, to clarify these inconsistencies and help drive research in the area forward. This paper presents our vision for a novel framework that encapsulates the automated detection of critical software issues in the system through the prediction of functional defects and security vulnerabilities, automating their mitigation through patch generation, as well as automating the propagation of these patches to systems in production. Our research agenda aggregates existing work from different software engineering fields which we hope will facilitate collaboration between the communities and encourage new research directions. In our future work, we plan to begin realizing this framework.

References

[1] A. Nilizadeh and G. T. Leavens, "Be realistic: Automated program repair is a combination of undecidable problems," *Proceedings - International Workshop on Automated Program Repair, APR 2022*, pp. 31–32, 2022.

[2] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, pp. 1165–1178, 2008.

[3] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," in *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings* (K. Havelund, G. J. Holzmann, and R. Joshi, eds.), vol. 9058 of *LNCS*, pp. 3–11, Springer, 2015.

[4] A. Dasso and A. Funes, "Verification, validation and testing in software engineering," *Verification, Validation and Testing in Software Engineering*, pp. 1–428, 2006.

[5] E. W. Dijkstra, "Notes on structured programming," in *NATO Software Engineering Conference*, 1969.

[6] C. Jones, O. Bonsignour, and J. Subramanyam, *The Economics of Software Quality*. Addison-Wesley, 2011.

[7] M. Gómez, B. Adams, W. Maalej, M. Monperrus, and R. Rouvoy, "App store 2.0: From crowdsourced information to actionable feedback in mobile ecosystems," *IEEE Software*, vol. 34, pp. 81–89, 2017.

[8] X. Zhang, Y. Xu, S. Qin, S. He, B. Qiao, Z. Li, H. Zhang, X. Li, Y. Dang, Q. Lin, M. Chintalapati, S. Rajmohan, and D. Zhang, "Onion: Identifying incident-indicating logs for cloud systems," *ESEC/FSE 2021 - Proceedings of the 29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, vol. 21, pp. 1253–1263, 2021.

[9] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou, "Triage: Diagnosing production run failures at the user's site," *Operating Systems Review (ACM)*, pp. 131–144, 2007.

[10] F. Araujo and T. Taylor, "Improving cybersecurity hygiene through jit patching," *ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1421–1432, 2020.

[11] L. Zhou, F. Zhang, J. Liao, Z. Ning, J. Xiao, K. Leach, W. Weimer, and G. Wang, "Kshot: Live kernel patching with smm and sgx," *Proceedings - 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020*, pp. 1–13, 2020.

[12] M. Russinovich, N. Govindaraju, M. Raghuraman, D. Hepkin, J. Schwartz, and A. Kishan, "Virtual machine preserving host updates for zero day patching in public cloud," *EuroSys 2021 - Proceedings of the 16th European Conference on Computer Systems*, vol. 21, pp. 114–129, 2021.

[13] T. Hynninen, J. Kasurinen, A. Knutas, and O. Taipale, "Software testing: Survey of the industry practices," *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2018 - Proceedings*, pp. 1449–1454, 2018.

[14] M. K. Thota, F. H. Shajin, and P. Rajesh, "Survey on software defect prediction techniques," *International Journal of Applied Science and Engineering*, vol. 17, pp. 331–344, 2020.

[15] Z. Shen and S. Chen, "A survey of automatic software vulnerability detection, program repair, and defect prediction techniques," *Security and Communication Networks*, 2020.

[16] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, pp. 56–65, 2019.

[17] Z. Liu, C. Chen, A. Ejaz, D. Liu, and J. Zhang, "Automated binary analysis: A survey," in *Algorithms and Architectures for Parallel Processing - 22nd International Conference, ICA3PP 2022, Copenhagen, Denmark, October 10-12, 2022, Proceedings* (W. Meng, R. Lu, G. Min, and J. Vaidya, eds.), vol. 13777 of *Lecture Notes in Computer Science*, pp. 392–411, Springer, 2022.

[18] M. K. Gupta and P. Chandra, "A comprehensive survey of data mining," *International Journal of Information Technology (Singapore)*, vol. 12, pp. 1243–1257, 2020.

[19] C. Islam, V. Prokhorenko, and M. A. Babar, "Runtime software patching: Taxonomy, survey and future directions," *Journal of Systems and Software*, vol. 200, 2022.

[20] V. Nowack, D. Bowes, S. Counsell, T. Hall, S. Haraldsson, E. Winter, and J. Woodward, "Exploiting fault localisation for efficient program repair," *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pp. 311–312, 2020.

[21] M. Harman, "Scaling genetic improvement and automated program repair," in *3rd IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2022, Pittsburgh, PA, USA, May 19, 2022*, pp. 1–7, IEEE, 2022.

[22] A. Saieva and G. Kaiser, "Binary quilting to generate patched executables without compilation," *FEAST 2020 - Proceedings of the 2020 ACM Workshop on Forming an Ecosystem Around Software Transformation*, pp. 3–8, 2020.

[23] A. Popovici, G. Alonso, and T. R. Gross, "Just-in-time aspects: efficient dynamic weaving for java," in *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, AOSD 2003, Boston, Massachusetts, USA, March 17-21, 2003* (W. G. Griswold and M. Aksit, eds.), pp. 100–109, ACM, 2003.

[24] T. A. Limoncelli, "The small batches principle," *Queue*, vol. 14, 2016.

[25] A. Truelove, E. S. de Almeida, and I. Ahmed, "We'll fix it in post: What do bug fixes in video game update notes tell us?," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pp. 736–747, IEEE, 2021.

[26] R. Pierce, "Parallel design and development for documentation projects," *Communication Design Quarterly Review*, vol. 6, pp. 6–8, 2005.

[27] K. S. Trivedi, R. Mansharamani, D. S. Kim, M. Grottke, and M. Nambiar, "Recovery from failures due to mandelbugs in it systems," *Proceedings of IEEE Pacific Rim International Symposium on Dependable Computing, PRDC*, pp. 224–233, 2011.

[28] T. Aurisch and A. Jacke, "Handling vulnerabilities with mobile agents in order to consider the delay and disruption tolerant characteristic of military networks," *2018 International Conference on Military Communications and Information Systems, ICMCIS 2018*, pp. 1–7, 2018.

[29] N. Kerzazi, "Branching strategies based on social networks," *2013 1st International Workshop on Release Engineering, RELENG 2013 - Proceedings*, pp. 25–28, 2013.

[30] F. Daniel, E. Arvid, and H. Roland, "How mobile app design overhauls can be disastrous in terms of user perception," *ACM Transactions on Social Computing*, vol. 3, pp. 1–21, 2020.

[31] S. Casco, "Securing windows and iis," *Hack Proofing ColdFusion*, pp. 261–336, 2002.

[32] M. Gupta, S. Banerjee, M. Agrawal, and H. R. Rao, "Security analysis of internet technology components enabling globally distributed workplaces-a framework," *ACM Transactions on Internet Technology (TOIT)*, vol. 8, 2008.

[33] A. Agarwal and N. K. Garg, "Effective test strategy model for ensuring ftr of hot-fix," in *ICROIT 2014 - Proceedings of the 2014 International Conference on Reliability, Optimization and Information Technology*, pp. 40–43, IEEE Computer Society, 2014.

[34] J. Stanger, "Securing Windows 2000 advanced server and Red Hat Linux 6 for e-mail services," *E-Mail Virus Protection Handbook*, pp. 295–332, 2000.

[35] J. Xing, Y. Qiu, K. F. Hsu, H. Liu, M. Kadosh, A. Lo, A. Akella, T. Anderson, A. Krishnamurthy, T. S. Ng, and A. Chen, "A vision for runtime programmable networks," *HotNets 2021 - Proceedings of the 20th ACM Workshop on Hot Topics in Networks*, pp. 91–98, 2021.

[36] M. Vögler, J. M. Schleicher, C. Inzinger, and S. Dustdar, "A scalable framework for provisioning large-scale iot deployments," *ACM Transactions on Internet Technology (TOIT)*, vol. 16, 2016.

[37] S. V. Karale and V. Kaushal, "An automation framework for configuration management to reduce manual intervention," *ACM International Conference Proceeding Series*, vol. 12-13-August-2016, 2016.

[38] C. Bailey, "Configuring windows 2000 without active directory," *Assembly*, vol. 44, p. 79, 2001.

[39] N. Weichbrodt, T. B. G. J. Heinemann, T. B. G. L. Almstedt, R. Kapitza, T. B. Germany, J. Heinemann, L. Almstedt, and P.-L. Aublin, "Sgx-dl: Dynamic loading and hot-patching for secure applications: Experience paper," *Middleware 2021 - Proceedings of the 22nd International Middleware Conference*, pp. 91–103, 2021.

[40] C. Reffett and D. Fleck, "Securing applications with dyninst," *2015 IEEE International Symposium on Technologies for Homeland Security, HST 2015*, 2015.

[41] S. Biedermann, S. Katzenbeisser, and J. Szefer, "Hot-hardening: Getting more out of your security settings," *ACM International Conference Proceeding Series*, vol. 2014-December, pp. 6–15, 2014.

[42] E. M. Rudd, A. Rozsa, M. Günther, and T. E. Boult, "A survey of stealth malware attacks, mitigation measures, and steps toward autonomous open world solutions," *IEEE Communications Surveys and Tutorials*, vol. 19, pp. 1145–1172, 2017.

[43] Y. Shao, R. Wang, X. Chen, A. M. Azab, and Z. M. Mao, "A lightweight framework for fine-grained lifecycle control of android applications," *Proceedings of the 14th EuroSys Conference 2019*, vol. 19, 2019.

[44] S. Han, D. Baby, and V. Mendelev, "Residual adapters for targeted updates in rnn-transducer based speech recognition system," in *IEEE Spoken Language Technology Workshop, SLT 2022, Doha, Qatar, January 9-12, 2023*, pp. 160–166, IEEE, 2022.

[45] F. Hargrave, *Hargrave's Communications Dictionary*. IEEE, 2010.

[46] A. K. Sood, S. Zeadally, and R. Bansal, "Exploiting trust: Stealthy attacks through socioware and insider threats," *IEEE Systems Journal*, vol. 11, pp. 415–426, 2017.

[47] K. Chen, Y. Li, Y. Chen, C. Fan, Z. Hu, and W. Yang, "Glib: Towards automated test oracle for graphically-rich applications," *ESEC/FSE 2021 - Proceedings of the 29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, vol. 21, pp. 1093–1104, 2021.

[48] H. Oosterhuis and M. D. D. Rijke, "Robust generalization and safe query-specializationin counterfactual learning to rank," *The Web Conference 2021 - Proceedings of the World Wide Web Conference, WWW 2021*, pp. 158–170, 2021.

[49] C. Farnell, E. Soria, J. Jackson, and H. A. Mantooth, "Cyber protection of grid-connected devices through embedded online security," *2021 IEEE Design Methodologies Conference, DMC 2021*, 2021.

[50] T. Illes-Seifert and B. Paech, "Exploring the relationship of a file's history and its fault-proneness: An empirical study," *Proceedings - Testing: Academic and Industrial Conference Practice and Research Techniques, TAIC PART 2008*, pp. 13–22, 2008.

[51] Y. Chen, Y. Li, L. Lu, Y. Lin, H. Vijayakumar, Z. Wang, and X. Ou, "Instaguard: Instantly deployable hot-patches for vulnerable system programs on android," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, The Internet Society, 2018.

[52] G. Candea and P. Godefroid, "Automated software test generation: Some challenges, solutions, and recent advances," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10000, pp. 505–531, 2019.

[53] I. Mesecan, D. Blackwell, D. Clark, M. B. Cohen, and J. Petke, "Hypergi: Automated detection and repair of information flow leakage," in *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pp. 1358–1362, IEEE, 2021.

[54] O. Tunde-Onadele, Y. Lin, J. He, and X. Gu, "Toward just-in-time patching for containerized applications," in *Proceedings of the 7th Annual Symposium on Hot Topics in the Science of Security, HotSoS 2020, Lawrence, Kansas, USA, September 22-24, 2020* (P. Alexander, D. Davidson, and B. Choi, eds.), pp. 30:1–30:2, ACM, 2020.

[55] J. Bard, S. Jacobs, and Y. Vizel, "Automatic and incremental repair for speculative information leaks." https://arxiv.org/abs/2305.10092, 2023.

[56] A. Saieva and G. Kaiser, "Update with care: Testing candidate bug fixes and integrating selective updates through binary rewriting," *Journal of Systems and Software*, vol. 191, p. 111381, 2022.

[57] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, 2019.

[58] C. L. Goues, T. V. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, pp. 54–72, 2012.

[59] Y. Yuan and W. Banzhaf, "Arja: Automated repair of java programs via multi-objective genetic programming," *IEEE Transactions on Software Engineering*, 2018.

[60] M. Motwani, M. Soto, Y. Brun, R. Just, and C. L. Goues, "Quality of automated program repair on real-world defects," *IEEE Transactions on Software Engineering*, vol. 48, pp. 637–661, 2022.

[61] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of chatgpt," in *IEEE/ACM International Workshop on Automated Program Repair*, 2023.

[62] A. Bartel, J. Klein, M. Monperrus, and Y. L. Traon, "Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot," *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis, SOAP 2012*, pp. 27–38, 2012.

[63] Z. Xu, Y. Zhang, L. Zheng, L. Xia, C. Bao, Z. Wang, and Y. Liu, "Automatic hot patch generation for android kernels," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020* (S. Capkun and F. Roesner, eds.), pp. 2397–2414, USENIX Association, 2020.