Static analysis-based approaches for Secure Software Development

Miltiadis Siavvas[1,2], Erol Gelenbe[1], Dionysios Kehagias[2], and Dimitrios Tzovaras[2]

[1] Imperial College London, SW7 2AZ, London, United Kingdom
m.siavvas16@imperial.ac.uk, e.gelenbe@imperial.ac.uk
[2] Centre for Research and Technology Hellas, Thessaloniki, Greece
diok@iti.gr,dimitrios.tzovaras@iti.gr

Abstract. Software security is a matter of major concern for software development enterprises that wish to deliver highly secure software products to their customers. Static analysis is considered one of the most effective mechanisms for adding security to software products. The multitude of static analysis tools that are available provide a large number of raw results that may contain security-relevant information, which may be useful for the production of secure software. Several mechanisms that can facilitate the production of both secure and reliable software applications have been proposed over the years. In this paper, two such mechanisms, particularly the vulnerability prediction models (VPMs) and the optimum checkpoint recommendation (OCR) mechanisms, are theoretically examined, while their potential improvement by using static analysis is also investigated. In particular, we review the most significant contributions regarding these mechanisms, identify their most important open issues, and propose directions for future research, emphasizing on the potential adoption of static analysis for addressing the identified open issues. Hence, this paper can act as a reference for researchers that wish to contribute in these subfields, in order to gain solid understanding of the existing solutions and their open issues that require further research.

Keywords: Software Security, Reliability, Static Analysis, Vulnerability Prediction, Checkpoint and Restart

1 Introduction

Software security is usually considered an afterthought in the software development lifecycle (SDLC). It is normally added after the implementation of software products chiefly by using mechanisms aiming to prevent malicious individuals from exploiting existing vulnerabilities (e.g. intrusion detection systems). However, the increasing number of the security incidents reported annually indicates the inability of these mechanisms to fully protect the software products against attacks [1, 2]. To this end, software houses have shifted their focus towards building software products that are highly secure (i.e. as vulnerability-free as possible) from the ground up.

A software vulnerability is a weakness in the specification, development, or configuration of software such that its exploitation can violate a security policy [3]. Most of the software vulnerabilities stem from a small number of common programming errors [4]. These errors are introduced by the developers during the coding phase, mainly due to their lack of security expertise [5], or due to the accelerated production cycles [6]. However, it is unrealistic to expect from them to remember thousands of security-related bug patterns and bad practices that they should avoid. As a result, efficient tools are required to help them avoid the introduction of such security bugs, and therefore write more secure code [7, 8].

Automatic static analysis (ASA) tools have been proven effective in uncovering security-related bugs early enough in the software development process [4]. Their main characteristic is that they are applied directly to the source or compiled code of the system, without requiring its execution [5]. However, since their results comprise long lists of raw warnings (i.e. alerts) or absolute values of software metrics, they do not provide real insight to the stakeholders of the software products. In fact, a great number of ASA tools have been proposed over the years providing a huge volume of such raw data, which may contain security-relevant information that may be useful for secure software development. Hence, appropriate knowledge extraction tools are needed on top of the raw results produced by ASA tools for facilitating the production of secure software.

To this end, the outputs of ASA tools have recently started being used for vulnerability prediction. In fact, recent studies have highlighted the ability of ASA tools to predict software components that contain vulnerabilities (i.e. vulnerable components) [9, 10]. The prediction outcome may provide useful insights to project managers about where to focus their testing efforts. However, none of the already proposed vulnerability prediction models (VPMs), especially those that are not based on static analysis, managed to achieve a satisfactory trade-off among accuracy, practicality and performance [10]. The accuracy of the existing VPMs in cross-project prediction is also observed to be poor. In addition to this, static analysis results can be used to facilitate the development of software products that are fault tolerant, and therefore reliable, from the beginning, since they can be used in order to highlight expensive loops and fault-prone components prior of which application-level checkpoints should be inserted. Although several mechanisms have been proposed for assisting developers in the selection and insertion of application-level checkpoints (e.g. [11, 12])), they failed to provide complete recommendations, while the required development effort is high.

To sum up, we can state that the current trend in the field of Software Security is the development of knowledge-discovery mechanisms for the intelligent processing of the results produced by ASA tools to support the secure software development. A thorough literature review led us to the conclusion that the exploitation of the raw results produced by ASA tools in order to conduct (i) prediction of vulnerable software components, and (ii) optimum checkpoint recommendation constitute two interesting subfields with potential positive impact on the production of secure and reliable software products. To this end, the purpose of this paper is to review the most significant attempts in each one

of the aforementioned subfields, identify existing open issues of high interest, and potentially propose direction for future research, emphasizing on how these mechanisms may leverage from static analysis. Hence, this paper can act as a reference for researchers that wish to contribute in these subfields, in order to gain a solid understanding of existing solutions and identify open issues that require further research. All these are presented in detail in the rest of the paper.

2 Literature Review

2.1 Vulnerability Prediction Modeling

Vulnerability prediction modeling is a subfield of software security, aiming to predict software components that are likely to contain vulnerabilities (i.e. vulnerable components). Vulnerability prediction models (VPMs) are normally built based on machine learning techniques that use software attributes as input, to discriminate between vulnerable and neutral components. These models can be used for prioritizing testing and inspection efforts, by allocating limited test resources to potentially vulnerable parts. Although it is a relatively new area of research, a great number of VPMs has already been proposed in the related literature. As stated in [9], the main VPMs that can be found in the literature utilize software metrics [13–22], text mining [23–28], and security-related static analysis alerts [29–32, 10] to predict vulnerabilities. These types of VPMs are analyzed in the rest of this section.

Software Metrics Shin and Williams [13, 14] were the first to investigate the ability of software metrics, particularly complexity metrics, to predict vulnerabilities in software products. Several regression models were built based on different subsets of the studied metrics in order to discriminate between vulnerable and non-vulnerable (i.e. neutral) functions. The results of their analysis (which was based on Mozilla JavaScript Engine) suggest that complexity metrics can be used only as weak indicators of software vulnerabilities. Based on the same code base, Nguyen and Tran [15] evaluated the ability of semantic complexity in vulnerability prediction. The models which were built using semantic complexity demonstrated on average a better predictive performance compared to the best models of [14]. In [16] an empirical study conducted on Mozilla Firefox and Whiteshark revealed the ability of execution complexity metrics (i.e. complexity metrics that are collected during the code execution) to discriminate between vulnerable and neutral functions. In particular, VPMs built on these features were found to predict vulnerability-prone functions with similar prediction performance to commonly used statically collected complexity metrics, but with lower inspection effort.

The main purpose of the previous studies was to empirically evaluate the experts' opinion that software complexity is the enemy of software security. The weak relationship that was generally observed between complexity and vulnerabilities led to the need for incorporating additional metrics in vulnerability prediction. Towards this end, Chowdhury and Zulkernine [17], based on 52 releases

of Mozilla Firefox, highlighted the ability of complexity, coupling, and cohesion (CCC) metrics to indicate the existence of vulnerabilities in software files. Based on this observation, the same authors proposed a framework for the automatic prediction of vulnerable files based on CCC metrics [18]. The VPMs that were built demonstrated a high accuracy and tolerable false positive rate. Shin et al. [19] examined the ability of complexity, code churn, and developer activity to discriminate between vulnerable and neutral files in two widely-used software products, namely Mozilla Firefox and Red Hat Linux Kernel. The results of the analysis suggest that the selected metrics may be used as sufficient indicators of vulnerabilities in software files, while those retrieved from software history may be stronger indicators than the commonly-used complexity measurements.

Moshtari et al. [20], contrary to previous studies, examined and highlighted the ability of software complexity to predict vulnerabilities between software products (i.e. cross-project prediction), based on 5 open-source software products, namely Mozilla Firefox, Linux Kernel, Apache Tomcat, Eclipse, and Open SCADA. Similarly, in a recent study [21], the predictive power of complexity and coupling in cross-project prediction was compared. The results revealed that complexity metrics had better predictive power than coupling metrics in cross-project prediction, while the combination of traditional complexity measurements with a newly proposed set of coupling metrics led to an improvement in the recall of the best complexity-based VPM that was built in this work.

Text Mining Apart from software metrics that have received much attention in the field of vulnerability prediction, VPMs using text mining have also demonstrated highly promising results. In this approach, the source code of the software artifacts is parsed and represented as a set of tokens (i.e. keywords). Subsequently, these tokens are intuitively combined and used to train vulnerability predictors. Neuhaus et al. [23] were the first to adopt a form of text mining for vulnerability prediction. They proposed Vulture, a VPM that predicts vulnerabilities in software components based on their import statements and function calls, which are parsed from their source code. An empirical evaluation of the model on Mozilla Firefox and Thunderbird, revealed that the proposed VPM was able to predict half of all the existing vulnerable components, and about one third of all the predictions were correct. Vulture is also the first known VPM that can be found in the related literature.

A more complete text mining-based prediction approach was introduced by Hovsepyan et al. [24]. According to their technique, each software component is characterized as a series of text terms extracted from their source code along with their associated frequencies, which are then used to forecast whether each component is likely to contain vulnerabilities. An empirical evaluation on 19 versions of a large-scale Android application, revealed that their technique may be promising for vulnerability prediction, as the produced predictors achieved sufficient precision (85% on average) and recall (87% on average). Based on these preliminary results, the same authors conducted a more elaborate empirical study to investigate the validity of their approach [25]. In particular, several

VPMs using Naïve Bayes and Random Forest algorithms were constructed and evaluated on a code base of 20 large-scale Android applications. The evaluation results revealed that the predictive power of the proposed models is equal or even superior to what is achieved by state-of-the-art VPMs, which indicates that text mining can be used for the construction of satisfactory VPMs. However, the produced models performed poorly in cross-project prediction, which can be explained by the fact that their predictions are based on text terms, which are highly project-specific features.

Pang et al [26] proposed an improvement of the aforementioned technique [24, 25], by employing N-Gram analysis. According to their proposal, continuous sequences of tokens should be used instead of raw text features for predicting vulnerable components. An empirical evaluation based on 4 android applications (retrieved from the same code base with [25]), revealed that SVM predictors built based on N-Gram technique were able to predict vulnerable components with high accuracy, precision and recall. In a recent replication of their study [27], the same authors observed that the adoption of deep neural networks instead of SVM, can also lead to VPMs with highly satisfactory predictive performance. Despite their promising results, these techniques are too expensive in terms of memory and execution time, due to the nature of their features (i.e. long sequences of text terms), which restricts their practicality.

In [28] a sophisticated two-tier composite approach called VULPREDICTOR was proposed to predict vulnerable files. VULPREDICTOR [28] analyzes text features along with software metrics and is built on top of an ensemble of classifiers. VULPREDICTOR outperforms the state-of-the-art approaches proposed by Walden et al. [33], which use either software metrics or text features for vulnerability prediction, but not both. This indicates that the combination of software metrics with text mining may be promising for vulnerability prediction.

Static Analysis Limited attempts can be also found in the related literature regarding the ability of security-related static analysis alerts to predict the existence of vulnerabilities. The idea of using static analysis alerts for the identification of attack- or vulnerability-prone software components was inspired by Gegick and Williams [29]. Based on this concept Gegick et al. [30] constructed several VPMs using ASA alerts density, code churn, and lines of code as inputs, while different combinations of these features were also considered. Recursive partitioning and logistic regression was employed for the construction of these models. An empirical evaluation of the produced VPMs on a large commercial telecommunication software system revealed that the model based on ASA alerts density and code churn was the best predictor, being able to detect 100% of the attack-prone components, with 8% false positive rate (FPR). This indicates that ASA alerts can be used effectively for vulnerability prediction, while their combination with other vulnerability indicators (e.g. software metrics) may also be promising. In [31] a replication of the work presented in [30] on a different code base (i.e. a large Cisco software system) led to different observations. In fact, the FPR of the produced model was found to be higher, which suggests that

the selection of the code base may influence the predictive performance of the produced models.

Moreover, several recent studies have highlighted the need for refining existing VPMs by using security-specific metrics and ASA alerts as inputs in order to improve their accuracy [32]. To this end, Yang et al. [10] proposed a novel VPM that uses security-specific warnings produced by an ASA tool to predict vulnerable components. The evaluation results revealed that the proposed approach may lead to an improvement of up to 5% in terms of accuracy compared to the state-of-the-art models proposed in [33] . This suggests that the adoption of security-specific warnings, and especially their combination with software metrics, may be beneficial for the production of accurate VPMs.

Comparison of Existing Models Finally, different empirical studies have shown that text mining-based models exhibit better predictive performance in comparison to other state-of-the-art techniques [33, 34, 9]. However, they perform poorly in cross-project prediction, which indicates that they are highly project-specific [33], while excessive amount of time and memory is required for their construction and regular application [9, 34]. Hence, VPMs that use software metrics (such as complexity, code churns etc.) and density of ASA alerts may be a more viable solution in practice [34], as they are less expensive to build and apply [34], and they perform slightly better in cross-project prediction [33]. This highly suggests that the adoption of security-related ASA alerts and statically collected software metrics may be a promising approach for the construction of more accurate, as well as practical VPMs. Hence, future research attempts should focus towards this direction.

Open Issues and Contributions Despite the multitude of VPMs that have been already proposed over the years, there are still many open issues that require further investigation. First of all, none of the already proposed techniques managed to achieve a satisfactory trade-off among accuracy, practicality and performance [10]. Accurate VPMs usually provide predictions at the binary level (i.e. file or component level), which is impractical in terms of inspection time, as binaries normally contain hundreds of source files [32]. More practical VPMs that provide predictions at the source code level are usually inaccurate [10] or they produce a large number of false positives (i.e. clean files wrongly predicted to be vulnerable), which renders the inspection process time-consuming and effort-demanding. On the contrary, models that are both accurate and practical, such as text mining-based VPMs [25], are highly expensive to build and apply [25, 34]. Thus, a model able to achieve a sufficient compensation among the previously mentioned factors is necessary.

Another issue is that the datasets used in the literature for the derivation of VPMs are constructed based chiefly on reported vulnerabilities of real products. However, not all of the vulnerabilities that a product contains are always reported, and therefore many components that are considered clean in the dataset may in fact be vulnerable. Moreover, the number of vulnerable files that

a software product includes is often too small [35], leading to highly imbalanced datasets, which influence significantly the accuracy of the produced predictors [32]. The usage of a highly balanced and sound dataset is expected to improve the accuracy of the produced VPMs. For this purpose, well-known vulnerability code bases like the Juliet suite [36] can be used for the construction of such a dataset.

The last issue is that the existing VPMs perform poorly in cross-project prediction [10]. This is normally due to the fact that they are based on project-specific features for providing their predictions. For instance, as stated previously, text mining VPMs [25] base their prediction on the frequencies of specific text features (i.e. keywords) extracted from the source code of software products, which makes them highly project-specific [34, 33]. The usage of security-related software metrics and alerts produced by ASA tools are expected to lead to more generic VPMs (i.e. models with sufficient cross-project performance), as these factors can catch more high-level and abstract attributes of software products.

To sum up, an interesting topic would be to investigate whether the combination of security-specific static analysis alerts and statically collected software metrics, along with the usage of a highly balanced and sound dataset, may lead to a VPM that achieves an acceptable trade-off among accuracy, practicality and performance. Another topic that worths examination is whether such a model demonstrates sufficient prediction performance in cross-project prediction.

2.2 Optimum Checkpoint Recommendation

As will be discussed in the present section, the application-level checkpoint and restart (ALCR) mechanism is the most effective mechanism for building software applications that are fault tolerant from the beginning [37–39]. However, since it is based on the deliberate insertion of checkpoints into the source code, it requires significant expertise and development effort. The optimum checkpoint recommendation (OCR) corresponds to the selection of the optimum source code locations where application-level checkpoints should be inserted, as well as of the optimum checkpointing frequency in case of repetitive processes (e.g. loops). This mechanism is highly useful during the SDLC as it helps developers make informed decisions regarding the optimum placement of the checkpoints, leading to more fault-tolerant and, thus, reliable software applications. In addition, by allowing the automatic insertion of the recommended checkpoints the development effort associated with the ALCR mechanism is reduced, and the developers' productivity remains almost unaffected. In the following sections, the related literature along with the fundamental background concepts of the overall field is provided.

Transaction-oriented Systems and Optimum Checkpoint Interval Checkpoint and rollback/recovery is one of the most widely-used mechanisms for adding fault tolerance to software applications [37–39]. It was originally developed for enhancing the reliability of transaction-oriented computer systems

(e.g. database or file systems), which are responsible for the sequential processing of incoming transactions [40]. If no fault tolerance mechanism is adopted by these systems, all the transactions need to be re-executed in case of a failure, leading to significant performance burden. According to the checkpoint and rollback/recovery scheme, at predetermined instants of times (i.e. intervals), a snapshot (i.e. a secure valid copy) of all the data that have been successfully processed so far is taken and stored in an area that cannot be affected by failures (e.g. a secondary file system) [40–42]. This stored snapshot is called checkpoint [40–42]. In case of a failure, a rollback/recovery is performed, during which the contents of the checkpoint are copied from the secondary memory to the main memory of the system [43] and all the transactions since the most recent checkpoint are re-executed [40–42]. By using this scheme, the number of transactions that need to be re-executed is significantly reduced.

The major challenge of such systems is the selection of the *optimum checkpoint interval (OCI)*, that is, the time interval between two successive checkpoints that maximizes the system availability [40, 42, 43]. The term availability corresponds to the probability that the system is available for processing transactions [43, 44]. Hence, the vast majority of the research attempts in this field have focused chiefly on the selection of the optimum checkpoint interval (e.g. [45, 43, 44]), as well as on the impact that the checkpoint interval may have on other qualities of transaction-oriented systems (e.g. [40, 42]). The first attempt for determining the OCI was conducted by Young [45], who attempted to compute the optimum time interval between two successive checkpoints so as to achieve a satisfactory trade-off between the time required for the establishment of a checkpoint, and the time required for the system to recover from a failure. The author proposed a simple yet practical formula for the calculation of the OCI, in which the optimum interval depends on the mean time to failures, as well as on the time required for the creation of a checkpoint.

The selection of the OCI is crucial, as the cost of checkpointing is observed to be high if the checkpoints are frequent [46]. Based on this observation, in [40] the author investigated the impact that the selection of the OCI may have on the system performance. The results of the analysis revealed that the optimum value of the interval between two successive checkpoints that maximizes the system availability, does not optimize its performance as well. Therefore, the OCI should be calculated in a way that a sufficient compensation between the contradictory factors of availability and performance is achieved. Another important parameter of transaction-oriented systems is their response time. Hence, in [42] the authors examined the impact that the selection of the checkpoint interval may have on the availability and the response time of transaction-oriented systems. The results of their analysis revealed that the average response time depends highly on the checkpoint interval, and that the interval that minimizes the average response time is considerably different from the one that maximizes the system availability (i.e. the OCI). The authors also proposed a mathematical model for representing transaction-oriented systems that utilize the checkpoint and

rollback/recovery mechanism. This model can be used both for the calculation of the OCI and for the estimation of its impact on important performance measures.

In [43] Gelenbe et al. proved that the checkpoint interval should be deterministic in order to maximize the system availability, and that the OCI is a function of the system load. Based on these observations, the authors proposed a new formula for the calculation of the OCI that also considers the system load among other parameters. In [44] the same authors, in an attempt to further enhance the completeness of the OCI calculation, proposed a formula that takes time dependence into account. In particular, they showed that the OCI is a function of (i) the system load and (ii) the time-dependent failure rate. The results of their analysis also highlighted that the checkpoint interval should be deterministic in order to maximize the availability of the system, further supporting their previous observation [43].

Finally, Gelenbe et al. [46, 47] proposed a new fault tolerant mechanism, called failure (or virus) tests, which can be used in conjunction to the traditional checkpoint rollback/recovery technique for further enhancing the reliability of transaction-oriented systems. According to their newly proposed approach, the data and the transaction trail of the system are periodically checked for errors and inconsistencies. If at least one error (or inconsistency) is detected, the system is forced to go through a recovery as if a failure occurred. The authors also proposed a method for the calculation of the OCI, as well as the optimum interval between two successive failure tests that maximizes the system availability.

Long-running Software Applications Although the idea of the checkpoint and rollback/recovery has been initially proposed for transaction-oriented systems, it has been also found promising for enhancing the reliability of long-running software applications [48, 49]. Long-running software applications are considerably more complex compared to the transaction-oriented systems, since even the most simple software programs consist of a tremendous number of execution states [50]. Therefore, periodically saving only the successfully processed data of a software application is not enough for ensuring its reliability. On the contrary, a "safe copy" (i.e. a checkpoint) of the overall execution state of the application should be taken and saved in a secondary file system that cannot be tampered by failures [37, 51]. This safe state can be used for recovering the execution of the program in case of a failure.

It should be noted that the majority of software failures are caused by design and implementation errors [50]. However, due to the high complexity of modern software products, it is impossible to guarantee their correctness, even with the most exhaustive validation and verification. Hence, since software applications are inevitably bundled with such issues, effective fault tolerance mechanisms are required to enhance their reliability [50, 52]. Several rollback/recovery-based fault tolerance mechanisms have been proposed over the years for enhancing the reliability of long-running software applications , including: (i) the Recovery Block (RB) scheme [50], (ii) the N-Version Programming (NVP) [53], and (iii) the Checkpoint and Restart (CR) [49] technique.

Despite their benefits in enhancing the reliability of software applications, NVP and RB approaches are characterized by high development costs, since they require the division of the source code into individual blocks and the definition of alternative implementations for each one of these blocks. Therefore, due to these high development costs, as well as to the significant overheads they introduce, their adoption is restricted to very critical applications (e.g. safety-critical applications), in which the reliability is the most important factor [54].

Checkpoint and Restart (CR) is a fault tolerance mechanism widely used for enhancing the reliability of long-running software applications [49, 39, 38], since it introduces significantly less overheads and development costs, compared to the aforementioned rollback/recovery fault tolerance mechanisms (e.g. [50, 53]). A CR mechanism is responsible for keeping a "safe copy" of the current execution state of the software application, and use this "safe copy" for restoring the application in case of a failure. According to [39], three types of CR exist, which are (i) the system-level CR, (ii) the library-level CR, and (iii) the application-level CR, each one of them having its own strengths and shortcomings.

Both system- and library-level CR techniques are reactive approaches for adding fault tolerance, since they allow checkpointing of software applications without requiring any modification to their source code. However, their major shortcoming is that they create checkpoints with large memory footprints, as the entire execution state of the application and the operating system processes is saved, which inevitably contains redundant information. Two representative examples of system- and library-level CR tools are BLCR [55] and DMTCP [56] respectively.

When the CR is built within the application itself, it is called application-level CR (ALCR) [57, 39]. Unlike its counterparts, it necessitates changes to the source code of the applications in order to define (i) the locations of the checkpoints, (ii) the checkpointing frequency, and (iii) the data that should be checkpointed. Although it requires significant development effort, it is considered the most effective CR approach [37–39], as it allows the creation of checkpoints with smaller memory footprints, since the minimum amount of information required for restoring the application state is essentially saved. A great number of tools for implementing ALCR in software applications can be found in the related literature [57]. A more detailed description and comparison of these types of CR can be found in [57, 49].

Several CR tools and libraries are available for ensuring the reliability of single-process software applications, including the well-known: BLCR [55], and Condor [11]. However, the CR approach has recently become an attractive area of research due to our increasing reliance on long-running multi-process HPC applications. Such applications are characterized by expensive and time-consuming computations, and therefore excessive re-computation should be avoided in case of a failure [37]. For these applications, a distributed CR scheme should be employed, in which the checkpoints of the individual processes that constitute the parallel job should be effectively combined in order to create consistent recovery states of the overall parallel application.

The most common approach for incorporating the CR mechanism into the HPC applications is by integrating it into libraries that are required for the implementation of such applications, like OpenMPI (e.g. [58]), OpenCL (e.g. [12]), and OpenMP (e.g. [37, 51]). For instance, in [58] the authors extended the OpenMPI library, which is commonly used by HPC applications, in order to support the CR fault tolerance mechanism. Contrary to previous fault tolerant MPI implementations that were characterized by complicated and difficult to use interfaces [48], the proposed implementation manages to automate the process of constructing the global checkpoint of the parallel application, increasing in that way its usability. In [12], the authors proposed CheCL, a tool for incorporating CR into OpenCL applications, since common CR libraries fail to checkpoint processes that use OpenCL. The main advantage of CheCL is that it does not require the modification of the application source code. Instead, the proposed tool monitors the execution of the software application and all the API calls are forwarded to an API proxy, which stores all the information required for restoring OpenCL objects. The application, which is decoupled from OpenCL calls, is then checkpointed using the BLCR [55] CR library.

As already mentioned, ALCR is the most effective CR mechanism as it incurs minimum overhead compared to its counterparts, but it requires significant development effort, which hinders its adoption in practice. To this end, Rodriguez et al. [51] proposed CPPC, a tool for providing ALCR to message passing applications. The tool manages to reduce the manual effort required by the developers, as it identifies the safe points of the applications where checkpoints should be inserted (i.e. code locations with no inconsistencies), and automatically implements the checkpoints. In fact, it identifies long-running loops and automatically inserts checkpoints at the first safe point of each loop. Losada et al.[37] proposed an application-level checkpointing solution for hybrid MPI-OpenMP applications. In fact, the proposed solution is an extension of the CPPC [51] tool, which allows checkpointing of applications implemented using either MPI or OpenMP, in order to support hybrid MPI-OpenMP applications.

Shahzad et al. [38] proposed CRAFT, a library for incorporating the application level CR fault tolerance mechanism to software applications implemented in C++. Similarly to CPPC [51] and [37], the proposed library aims to reduce the development cost associated with the ALCR mechanism, by allowing the identification of expensive loops and the automatic insertion of the application-level checkpoints. In a recent research attempt, Arora et al. [39] proposed ITALC, a tool that helps the developers in semi-automatically re-engineering their applications to insert the code for the implementation of ALCR mechanism, without compromising their productivity. ITALC identifies hotspots (i.e. expensive loops or suspicious commands) where checkpoints can be inserted, and prompts the user in order to select which of those hotspots should be checkpointed.

Open Issues and Contributions As already mentioned, among the CR mechanisms, ALCR is considered the most effective, since it leaves the minimum memory footprint, but it requires significant development effort and expertise for its

implementation [38, 39]. Although several libraries for assisting and automating the insertion of application level checkpoints have already been proposed (e.g. [38, 39, 37, 51]), they are hindered by a set of shortcomings. Firstly, existing approaches provide incomplete recommendations, since although they identify hotspots for the insertion of the checkpoints (e.g. [39, 51]), these hotspots are restricted only to expensive loops. Apart from the expensive loops, failure-prone software artifacts (e.g. classes or methods) should be also identified and checkpoints should be inserted prior to their execution, in order to achieve quick recovery in case of failure. Moreover, in case of expensive loops, existing approaches do not provide recommendations regarding the optimum checkpointing frequency. Static analysis can be used to highlight existing failure-prone components and expensive loops that may require checkpointing. It can be also used to calculate both the logical complexity and the total cost of the application loops, information that can be used for the recommendation of a reasonable checkpointing frequency for the loops that may require checkpointing. Such features are expected to reduce the expertise required for the implementation of the checkpoints, as well as the checkpointing overhead, as both the locations and the frequency of the checkpoints will be optimally defined.

3 Conclusion and Future Work

In the present study, two commonly used mechanisms for enhancing the security and reliability of software products, namely the vulnerability prediction models (VPMs) and the optimum checkpoint recommendation (OCR) mechanisms were examined, by investigating their state-of-the-art. Through our study we identified some interesting open issues regarding the aforementioned mechanisms that can be potentially addressed through static analysis. In particular, none of the existing VPMs that have been proposed so far has managed to achieve a satisfactory trade-off among the contradictory factors of accuracy, practicality and performance, while their predictive performance in cross-project prediction is generally observed to be poor. In addition, although several libraries and tools for assisting developers in the selection and insertion of application-level checkpoints have been proposed, they have failed to provide complete recommendations, since they focus exclusively on expensive loops, while they do not provide any recommendation regarding their checkpointing frequency. Therefore, an interesting direction for future research is to investigate whether the results produced by static analysis tools, can be used in order to (i) construct better VPMs, and (ii) facilitate the optimum selection of application-level checkpoint locations and frequencies.

Acknowledgements This work is partially funded by the European Union's Horizon 2020 Research and Innovation Programme through SDK4ED project under Grant Agreement No. 780572.

References

1. Salini, P., Kanmani, S.: Survey and analysis on security requirements engineering. Computers and Electrical Engineering **38**(6) (2012) 1785–1797
2. McGraw, G.: On bricks and walls: Why building secure software is hard. Computers & Security **21**(3) (2002) 229–238
3. Krsul, I.: Software Vulnerability Analysis. PhD thesis, Department of Computer Sciences, Purdue University (1998)
4. Chess, B., McGraw, G.: Static analysis for security. Security & Privacy, IEEE **2** (2004) 76–79
5. McGraw, G.: Software Security: Building Security In. Addison-Wesley Prof. (2006)
6. Boehm, B., Basili, V.R.: Software Defect Reduction Top 10 List. Computer (2001)
7. Wurster, G., van Oorschot, P.C.: The developer is the enemy. NSPW '08: Proceedings of the 2008 Workshop on New Security Paradigms (2008) 89–97
8. Green, M., Smith, M.: Developers are Not the Enemy!: The Need for Usable Security APIs. IEEE Security and Privacy **14** (2016)
9. Jimenez, M., Papadakis, M., Traon, Y.L.: Vulnerability Prediction Models : A case study on the Linux Kernel. In: 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM). (2016) 1–10
10. Yang, J., Ryu, D., Baik, J.: Improving vulnerability prediction accuracy with Secure Coding Standard violation measures. 2016 International Conference on Big Data and Smart Computing, BigComp 2016 (2016) 115–122
11. Litzkow, M., Tannenbaum, T., Linvy, M.: Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report CS-TR-199701346, University of Wisconsin, Madison (1997)
12. Takizawa, H., Koyama, K., Sato, K., Komatsu, K., Kobayashi, H.: CheCL: Transparent checkpointing and process migration of OpenCL applications. Proceedings - 25th IEEE International Parallel and Distributed Processing Symposium (2011)
13. Shin, Y., Williams, L.: Is complexity really the enemy of software security? In: Proceedings - ACM Conference on Computer and Communications Security. (2008)
14. Shin, Y., Williams, L.: An empirical model to predict security vulnerabilities using code complexity metrics. In: Proceedings of the 2008 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. (2008)
15. Nguyen, V.H., Tran, L.M.S.: Predicting vulnerable software components with dependency graphs. Proceedings of the 6th International Workshop on Security Measurements and Metrics - MetriSec '10 (2010)
16. Shin, Y., Williams, L.: An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. In: Proceedings - International Conference on Software Engineering. (2011)
17. Chowdhury, I., Zulkernine, M.: Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? In: Proc. of the 2010 ACM Symposium on Applied Comp. (2010)
18. Chowdhury, I., Zulkernine, M.: Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. Journal of Systems Architecture **57** (2011)
19. Shin, Y., Meneely, A., Williams, L., Osborne, J.A.: Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. IEEE Transactions on Software Engineering **37**(6) (2011) 772–787
20. Moshtari, S., Sami, A., Azimi, M.: Using complexity metrics to improve software security. Computer Fraud and Security (2013)

21. Moshtari, S., Sami, A.: Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction. Proceedings - 31st Annual ACM Symposium on Applied Computing - SAC '16 (2016)
22. Alves, H., Fonseca, B., Antunes, N.: Software Metrics and Security Vulnerabilities: Dataset and Exploratory Study. Proceedings - 2016 12th European Dependable Computing Conference, EDCC 2016 (2016)
23. Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A.: Predicting vulnerable software components. Proceedings of the 14th ACM conference on Computer and communications security CCS 07 (2007) 529
24. Hovsepyan, A., Scandariato, R., Joosen, W., Walden, J.: Software vulnerability prediction using text analysis techniques. Proceedings of the 4th international workshop on Security measurements and metrics - MetriSec '12 (2012) 7
25. Scandariato, R., Walden, J., Hovsepyan, A., Joosen, W.: Predicting vulnerable software components via text mining. IEEE Transactions on Software Engineering **40**(10) (2014) 993–1006
26. Pang, Y., Xue, X., Namin, A.S.: Predicting Vulnerable Software Components through N-Gram Analysis and Statistical Feature Selection. In: 2015 IEEE 14th International Conference on Machine Learning and Applications. (2015)
27. Pang, Y., Xue, X., Wang, H.: Predicting Vulnerable Software Components through Deep Neural Network. Proceedings of the 2017 International Conference on Deep Learning Technologies - ICDLT '17 (2017) 6–10
28. Zhang, Y., Lo, D., Xia, X., Xu, B., Sun, J., Li, S.: Combining Software Metrics and Text Features for Vulnerable File Prediction. Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems, ICECCS **2016-Janua** (2016) 40–49
29. Gegick, M., Williams, L.: Toward the use of automated static analysis alerts for early identification of vulnerability- and attack-prone components. Second International Conference on Internet Monitoring and Protection, ICIMP 2007 (2007)
30. Gegick, M., Williams, L., Osborne, J., Vouk, M.: Prioritizing Software Security Fortification through Code-Level Metrics. Proceedings of the 4th ACM workshop on Quality of Protection (2008) 31–38
31. Gegick, M., Rotella, P., Williams, L.: Predicting attack-prone components. Proceedings - 2nd International Conference on Software Testing, Verification, and Validation, ICST 2009 (2009) 181–190
32. Morrison, P., Herzig, K., Murphy, B., Williams, L.: Challenges with Applying Vulnerability Prediction Models. Proceedings of the 2015 Symposium and Bootcamp on the Science of Security (2015) 4:1–4:9
33. Walden, J., Stuckman, J., Scandariato, R.: Predicting vulnerable components: Software metrics vs text mining. Proceedings - International Symposium on Software Reliability Engineering, ISSRE (2014) 23–33
34. Tang, Y., Zhao, F., Yang, Y., Lu, H., Zhou, Y., Xu, B.: Predicting Vulnerable Components via Text Mining or Software Metrics? An Effort-Aware Perspective. Proceedings - 2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015 (2015) 27–36
35. Shin, Y., Williams, L.: Can traditional fault prediction models be used for vulnerability prediction? Empirical Software Engineering **18**(1) (2013) 25–59
36. Boland, T., Black, P.E.: Juliet 1.1 C/C++ and java test suite. Computer (Long. Beach. Calif.) **45**(10) (2012) 88–90
37. Losada, N., Martín, M.J., Rodríguez, G., Gonzalez, P.: Portable application-level checkpointing for hybrid MPI-OpenMP applications. Proc. Comp. Science (2016)

38. Shahzad, F., Thies, J., Kreutzer, M., Zeiser, T., Hager, G., Wellein, G.: CRAFT: A library for easier application-level Checkpoint/Restart and Automatic Fault Tolerance. CoRR (2017)
39. Arora, R.: ITALC : Interactive Tool for Application - Level Checkpointing. Proceedings of the Fourth International Workshop on HPC User Support Tools (2017)
40. Gelenbe, E.: A Model of Roll-back Recovery with Multiple Checkpoints. In: Proceedings of the 2Nd International Conference on Software Engineering. ICSE '76, Los Alamitos, CA, USA, IEEE Computer Society Press (1976) 251–255
41. Gelenbe, E.: Model of information recovery using the method of multiple checkpoints. Automation and Remote Control **40**(4) (1979) 598–605
42. Gelenbe, E., Derochette, D.: Performance of Rollback Recovery Systems Under Intermittent Failures. Commun. ACM **21**(6) (1978) 493–499
43. Gelenbe, E.: On the Optimum Checkpoint Interval. Journal of the ACM **26**(2) (1979) 259–270
44. Gelenbe, E., Hernàndez, M.: Optimum Checkpoints With Age-Dependent Failures. Acta Informatica **531** (1990) 519–531
45. Young, J.W.: A First Order Approximation to the Optimum Checkpoint Interval. Commun. ACM **17**(9) (1974) 530–531
46. Gelenbe, E., Hernàndez, M.: Enhanced availability of transaction oriented systems using failure tests. Software Reliability Engineering, 1992. Proceedings., Third International Symposium on (1992) 342–350
47. Gelenbe, E., Hernández, M.: Virus tests to maximize availability of software systems. Theoretical Computer Science **125**(1) (1994) 131–147
48. Elnozahy, E.N., Alvisi, L., Wang, Y.M., Johnson, D.B.: A Survey of Rollback-recovery Protocols in Message-passing Systems. ACM Comput. Surv. (2002)
49. Egwutuoha, I.P., Levy, D., Selic, B., Chen, S.: A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. Journal of Supercomputing **65**(3) (2013) 1302–1326
50. Randell, B.: System Structure for Software Fault Tolerance. Science (2) (1975)
51. Rodríguez, G., Martín, M.J., González, P., Touriño, J., Doallo, R.: CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications. Concurrency and Computation: Practice and Experience **22**(6) (2010) 749–766
52. Carzaniga, A., Gorla, A., Perino, N., Pezzè, M.: Automatic Workarounds: Exploiting the Intrinsic Redundancy of Web Applications. Volume 24. (2015)
53. Chen, L., Avizienis, A.: N-version programming: A fault-tolerance approach to reliability of software operation. In: Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8). Volume 1. (1978) 3–9
54. Armoush, A., Salewski, F., Kowalewski, S.: A hybrid fault tolerance method for recovery block with a weak acceptance test. Proceedings of The 5th International Conference on Embedded and Ubiquitous Computing, EUC 2008 **1** (2008) 484–491
55. Duell, J., Hangrove, P., Roman, E.: The design and implementation of berkeley lab's linux checkpoint/restart. Berkeley Lab Technical Report (2002)
56. Ansel, J., Arya, K., Cooperman, G.: DMTCP: Transparent checkpointing for cluster computations and the desktop. IPDPS 2009 - Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium (2009)
57. Walters, J., Chaudhary, V.: Application-Level Checkpointing Techniques for Parallel Programs. Distributed Computing and Internet Technology (2006) 221–234
58. Hursey, J., Squyres, J.M., Mattox, T.I., Lumsdaine, A.: The Design and Implementation of Checkpoint / Restart Process Fault Tolerance for Open MPI . Architecture (2007)