POLITECNICO DI MILANO

Corso di Laurea Magistrale in Computer Science and Engineering

Scuola di Ingegneria Industriale e dell'Informazione

GitRDone: Enabling Fast Patch Propagation in Related Software Repositories

Relatore:
Prof. Stefano Zanero
Correlatore:
Aravind Machiry

Tesi di Laurea di:
Eric Camellini (836576)

Anno Accademico 2015-2016

Ampio Estratto

La diffusione del modello di sviluppo software open-source ha rappresentato una rivoluzione nel settore dell'informatica: incoraggia gli sviluppatori a collaborare in maniera aperta e libera, e ricerche recenti mostrano che è più sicuro [18, 54] della sua controparte closed-source. Tuttavia, fare in modo che le patch che vengono applicate ad un progetto open-source, sul suo repository principale, vengano applicate anche a tutti i repository connessi ad esso (ad esempio, a tutte le sue fork) è un problema serio [50]: le patch e le modifiche non possono essere applicate direttamente ad essi, dato che farlo potrebbe avere effetti imprevedibili sul codice di tali progetti, che con ogni probabilità si è diversificato da quello originale. In più, gestire le incompatibilità e le dipendenze tra le diverse parti di questi sistemi richiede sforzi considerevoli [51]. Si consideri, ad esempio, il sistema Android, che è costruito attorno ad una fork del kernel Linux: applicare patch provenienti dal repository principale del kernel su tale fork senza testarne scrupolosamente gli effetti potrebbe compromettere alcune funzionalità del sistema nella sua interezza.

Per risolvere il problema descritto, pur riuscendo a mantenere aggiornato il software, i maintainer e gli sviluppatori che lavorano su fork o progetti connessi ad un altro progetto devono scegliere accuratamente le patch e le modifiche che vogliono applicare. Questo processo richiede sforzi e tempo [6,56]: devono analizzare le patch a mano per identificare quelle da applicare, studiare il codice che modificano, capirne gli effetti, adattarle al codice del loro progetto e testare che tutto funzioni ancora come previsto dopo che vengono applicate. Per questi motivi, ci vuole tempo prima che le modifiche vengano propagate dal repository principale di un progetto a quelli correlati [52] (ad esempio, Android 7.1.1 è basato sulla versione 3.1 del kernel Linux, mentre l'ultima versione è la 4.9.8 [2]).

Se si considerano le patch di sicurezza, risolvere questo problema diventa

fondamentale: le modifiche relative alla sicurezza del software dovrebbero essere applicate il più velocemente possibile. Per far in modo che questo avvenga, sono stati creati dei database di vulnerabilità come quello contenente le CVE (Common Vulnerabilities and Exposures): tali database possono essere usati come riferimento dagli sviluppatori, in quanto contengono informazioni sulle più recenti falle di sicurezza che vengono identificate nei vari progetti. Questi database contengono anche riferimenti alle patch corrispondenti a tali falle: sono le patch che i maintainer dovrebbero applicare immediatamente sui progetti correlati a quello su cui sono state identificate le falle originali. Nonostante l'esistenza di questi database, le patch di sicurezza vengono tuttora applicate a fork e progetti correlati con un ritardo importante [10, 29, 38, 42]. Nel 2016, 80 vulnerabilità note (cioè, 80 CVE) sono state patchate sul sistema Android con più di un anno di ritardo [1]; questo significa che le falle corrispondenti sono rimaste nel sistema per tutto quel tempo. 76 di queste erano state scoperte e corrette sul repository principale dei corrispondenti progetti nel 2014, 2 nel 2013 e due nel 2012. Un altro problema di questi database è che gli sviluppatori potrebbero interpretare nel modo sbagliato la gravità di una falla per cui sviluppano una patch, e quindi non richiedere che venga inserita in uno di essi.

Gli approcci che sono stati ideati e sviluppati con lo scopo di facilitare questo processo di selezione e applicazione delle patch fanno affidamento su informazioni come i messaggi dei commit in un repository o le differenze riga per riga nel codice sorgente (cioè, il *diff*) [9, 49, 57]; altri approcci si basano sulla ricerca di pattern specifici [41]. Tutte queste tecniche sono leggere, veloci e possono essere applicate su progetti di tutte le dimensioni e complessità. Tuttavia, fanno affidamento su informazione che spesso non è affidabile, come i messaggi dei commit [3, 11, 53], e funzionano solo su patch di dimensioni semplici (questo perché tali tecniche non cercano di analizzare semanticamente i cambiamenti che una patch apporta, quindi non ne comprendono davvero gli effetti). Altri approcci provano invece a studiare le differenze nel codice a livello semantico utilizzando tecniche di analisi statica (cioè, static-analysis) [12,13,31,32,48] e, in particolare, esecuzione simbolica (cioè, symbolic execution) [15,23,37,46], ma soffrono di un problema differente: non sono applicabili su progetti grandi e complessi [17]. Altri approcci ancora, per funzionare, hanno bisogno del build environment del software analizzato [16], che significa che, in pratica, non è possibile utilizzarli su progetti con un numero elevato di configurazioni possibili (ad esempio, il kernel

Linux).

Una soluzione che faciliterebbe il processo di selezione e applicazione delle patch è creare un sistema capace di identificare quali sono quelle che potrebbero essere applicate senza bisogno di testare il software, dato che ne preservano le funzionalità originali. Tale sistema potrebbe essere usato per identificare questo tipo di patch sul repository principale di un progetto e successivamente applicarle in automatico su una fork di destinazione, o per notificare i maintainer dell'esistenza di tali patch. In questa ricerca abbiamo progettato e sviluppato una tecnica di analisi statica capace di determinare se una data patch su codice sorgente può essere applicata senza bisogno di essere testata scrupolosamente: questo significa che, molto probabilmente, tale patch preserva le funzionalità del software. Chiameremo questo tipo di patch *non dannose*. Successivamente, abbiamo implementato GitRDone, un programma basato su tale tecnica.

Il nostro programma, al contrario di quelli sviluppati in precedenti ricerche, fa affidamento soltanto sul codice sorgente del file che viene modificato, prima e dopo l'applicazione della patch analizzata (cioè, non ha bisogno di messaggi di commit, build environment ecc.), ed è anche, leggero, veloce e utilizzabile su progetti complessi. Lo abbiamo testato su 39,191 patch reali estratte da 10 differenti repository di progetti kernel ed è stato in grado di identificare 8,638 patch non dannose, con una precisione del 96.00%, mostrando quindi come potrebbe effettivamente essere utile per gli sviluppatori nel risolvere i problemi descritti sopra. Inoltre, lo abbiamo testato su 191 patch che correggono delle CVE e possiamo affermare che una porzione significativa delle patch di sicurezza fa parte di questa categoria di patch non dannose; questo significa che tale programma può essere utile anche per migliorare la sicurezza del software. In più, mostriamo anche come può essere usato per identificare delle patch di sicurezza per le quali non esiste una CVE corrispondente: abbiamo trovato diverse istanze reali di tale problema, dove le corrispondenti falle sono ancora presenti in alcuni progetti correlati a quelli su cui le abbiamo individuate. Rilasceremo il codice sorgente di GitRDone dopo la pubblicazione di questo lavoro.

Abstract

Despite the efforts of software maintainers, patches on open-source repositories are propagated from the main codebase to all the related projects (e.g., forks) with a significant delay. Previous work shows that this is true also for security patches, for which it represents a critical problem. Vulnerability databases (e.g., CVEs) were born to speed up the diffusion of critical patches; however, CVE patches are still applied with a delay and some security fixes lack corresponding CVE entries. Because of this, project maintainers could miss security patches when upgrading software, which is a huge problem.

In this paper we are the first to provide a definition of non-disruptive patches (*ndp*s). An *ndp* is a patch that should not disrupt the original functionality of the program, meaning that it can be applied with a minimal testing effort; we argue that most of the security fixes fall into this category. Furthermore, we show a technique to identify *ndp*s and implement GitRDone, a tool based on such a technique that works just by analyzing the source code of the original and patched versions of a file.

We run GitRDone on 39,191 patches, spanning over 10 different kernels repositories, and on 191 Android and Linux CVE patches. Results show that it can identify *ndp*s with 96.00% precision and that most of the CVE patches are *ndp*s. In addition, GitRDone identified patches that fix vulnerabilities that lack a CVE; 5 of these are still unpatched in different vendor kernels.

Acknowledgements

I would like to thank Professors Stefano Zanero and Federico Maggi for giving me the opportunity to go to the The Computer Security Group at UCSB for the internship during which I worked on this project, and for their expert advice and encouragement throughout my Master's degree. I would also like to thank Professors Giovanni Vigna and Christopher Kruegel for accepting me in their research laboratory and advising me during the time I spent there: doing research with them has been an honor and a privilege. Furthermore, I would like to thank all the people of the Seclab: they have been amazing coworkers and friends. In particular, special thanks go to Aravind Machiry, who supervised me throughout the whole experience, for his support and encouragement: without him, this project would neither have been started nor ever completed.

Contents

List of Figures

List of Tables

List of Abbreviations

IA Information Assurance

OSS Open-Source Software

OS Operating System

AST Abstract Syntax Tree

CFG Control Flow Graph

BB Basic Block

NDP Non-Disruptive Patch

CDG Control Dependency Graph

DDG Data Dependency Graph

UDG Use-Def Graph

API Application Programming Interface

Chapter 1

Introduction

The diffusion of the open-source software (OSS) model represented a revolution for the software industry: it encourages open and free collaboration, and prior research shows that it is also more secure [18, 54] than its closed source counterpart. However, patching all the repositories *related* to an OSS project (e.g., all its forks), starting from the patches applied on the main repository, is a major problem [50]: patches and changes cannot be applied just as they are because this would have unpredictable effects on the code bases of these related projects, that most likely diverged from the original one over time. Furthermore, dealing with incompatibilities and dependencies between the divergent parts of the systems requires a lot of effort [51] (e.g., Android relies on a Linux kernel fork: propagating patches from the kernel main repository on this fork without a complete and thorough testing could affect some Android features).

To solve the described problem and still keep software up to date, software developers and maintainers that work on projects related to another project need to accurately select and thoroughly test the patches and changes that they want to apply. This is a process that requires time and effort [6, 56]: they have to manually go through the changes to find the ones that need to be applied, analyze the code changed, understand the effects of these changes, adapt the patch to their code base and test that everything still works as expected after its application. For these reasons changes take a lot of time to propagate from the main repository of a project to the related ones [52] (e.g., Android 7.1.1 is built on the version 3.10 of the Linux kernel, while the latest one is version 4.9.8 [2]).

When considering security patches this problem becomes particularly critical: security-related fixes should be always applied as fast as possible. In order to ease and speed this process, vulnerability databases such as the CVE (Common Vulnerabilities and Exposures) were born: they can be used as a reference to remain up to date on the latest security issues that are identified on a certain project. These databases provide also references to the patches corresponding to a certain CVE, the ones that maintainers of related projects should immediately apply. However, vulnerability databases are not enough: security fixes are still applied on project forks with a significant delay [10, 29, 38, 42]. In the year 2016, 80 known vulnerabilities (i.e., CVEs) were patched on Android with more than one year delay [1], meaning that they remained unpatched in the Android code base for a significant amount of time. 76 of these were discovered and patched on the main repositories of their project in 2014, two in 2013 and two in 2012. Furthermore, developers can misinterpret the severity of a bug that they fix and fail to request a an entry in one of these vulnerability databases.

State-of-the-art approaches that can be used to ease the process of cherry-picking and applying changes rely information such as commit messages or source-code *diff* [9, 49, 57], while other approaches look for specific patterns [41]. All these techniques are lightweight, fast and scalable (i.e., suitable for large code bases). However, they rely on information that is often unreliable, like commit messages [3, 11, 53], and they work only on simple patches (i.e., without trying to semantically understand the source-code changes it is difficult to adapt an approach to complex patches). Other approaches try instead to study the semantic differences introduced by a change through static-analysis [12, 13, 31, 32, 48] and, specifically, symbolic execution [15, 23, 37, 46], but suffer from scalability issues [17]. Some other approaches make use of the build environment of the software [16], meaning that they cannot be applied, in practice, on highly configurable projects like the Linux kernel.

A solution that would ease the processes of selecting and applying patches and changes is to develop a system that can identifying the ones that is possible to apply without the need for subsequent testing, because they preserve the original functionality of the software. Such a tool could be used to identify this kind of patches on the main repository of a project and automatically apply them on a target fork, or to notify its maintainers about their existence. In this work we design a static analysis technique capable of determining if a given source code change could be applied with minimal testing: this means that it is most likely functionality-preserving. We call the patches identified by our technique *non-disruptive*. We then implement GitRDone, a tool based on this technique.

Our tool, in contrast with previous work, relies only on the source-code of a the updated file before and after the patch (i.e., it does not need commit messages, build environment, etc.), and it is also fast, lightweight and scalable. We evaluate it on 39,191 real commits spanning over 10 kernel projects and identify 8,638 non-disruptive patches with a 96.00% precision, showing how it can help developers in solving the problems described above. We then test it on 191 CVE patching commits and argue that a significant portion of all security-related fixes falls under this category of functionality-preserving patches, meaning that a tool like that would also improve the security of the software. Furthermore, we show how it can be used to identify security fixes for which there is no corresponding CVE: we found instances of these patches on the Linux kernel and we found some of these to be still unpatched on some of its forks. We will release the source code of GitRDone at the time of publication.

The outline of this work is as follows: in Chapter 2 we give an overview of the background and of the related work, and then explain in details the problem that we want to solve and our goals and contributions; in Chapter 3 we give a formal definition of non-disruptive patches, we describe a technique to identify them and show how GitRDone does it in practice (implementation details are shown in Chapter 4). Chapter 5, then, shows how we evaluated our tool, while in Chapter 6 we list its limitations, talk about possible future work and conclude.

Chapter 2

Motivation

2.1 Background

This section gives an overview of the concepts needed to understand the remaining part of this work; we only assume the reader to have a basic knowledge of the fundamentals of programming and computer science.

2.1.1 Software Security Overview

We now give an overview of the main software security concepts needed to understand what we do in this work.

2.1.1.1 Vulnerabilities and Exploits

A software defect (i.e., a bug) is called *vulnerability* if it allows to violate or reduce the software's *information assurance* (i.e., IA: protection of the user data and of its confidentiality, integrity and availability); any potential violation of the IA is called *threat*.

A way to use a vulnerability to cause unwanted behavior in the vulnerable software and to violate its IA is called *exploit*: the mere presence of a vulnerability does not always mean that the software can be violated, since there could be no way to exploit it. We call *attacker* whoever may attack the software (i.e., *intentionally* use one or more vulnerability exploits in order to violate its IA). Attackers are threats, according to our definition, but note that there could be also threats that are not attackers (e.g., too many clients talking to a server pose a threat to the software availability, but it is not their intention to do it).

2.1.1.2 Software Patching

A *patch* is an update designed to fix or improve a specific software; patches that fix one or more vulnerabilities or that improve the security of a software are called *security patches*. When the developers of a specific software become aware of a new vulnerability they should develop and distribute the corresponding security patch as soon as possible. One of the main problems of patching vulnerabilities is that not all the versions or instances of the vulnerable software are patched immediately (e.g., in case of Android applications, users could perform updates only once in a while and miss security fixes).

2.1.1.3 Vulnerability Disclosure

Computer security scientists and hackers agree on the fact that it is their social responsibility to notify the public about critical vulnerabilities and to disclose the corresponding details. However, vulnerability disclosure remains a debated topic, and different *disclosure policies* have been proposed:

- *Full disclosure*: After a vulnerability is discovered, its details are published as soon as possible, so that victims and attackers gain the same knowledge at the same time. The problem of this approach is that also the developers of the vulnerable software become aware of the vulnerability at the time of disclosure: this does not allow them to patch it in time.

- *Responsible disclosure*: Who discovers a vulnerability reports it to the vendor or maintainers and agrees with them on a period of time after which the vulnerability will be disclosed; this delay should allow the developers to patch it before it becomes public.

- *Non disclosure*: In this model, no information about vulnerabilities is disclosed whatsoever. This non-policy is applied by malicious hackers that have interest in selling exploits, or by vendors that think that information about vulnerabilities only helps the attackers.

Researchers and hackers also agree on the fact that information about vulnerabilities should be disclosed also when they are found directly by the vulnerable software vendor or maintainers: vulnerabilities should not be patched silently. However, once a vulnerability is disclosed, attackers become aware of it and can try to exploit it: this means that every unpatched instance of the software becomes potentially vulnerable to a possible publicly known exploit.

2.1.2 The Open-Source Software Model

The OSS model is a software development model that encourages open and free collaboration. The source code of an OSS project must be publicly available (e.g., the most common way to do it nowadays is to use a repository such as GitHub[1]); Given an OSS project, anyone is allowed to:

- Freely make use of it, following its licensing guidelines.
- Start a new project based on it; this practice is called *forking*, and the new code base is called *fork* (e.g., the Android OS is based on a *fork* of the Linux Kernel).
- Contribute to it (i.e., nowadays the most common way to do it is to fork the project, make changes and then submit a pull request that will then be reviewed by the maintainers).

Developers use version control systems such as `git`[2] to make changes to an open-source software project. Changes are usually applied locally (i.e., on the version of the project present only on the machine where changes are made) in batches, called *commits.* Sets of commits are then *pushed* on the public source-code of the project, often through the *pull requests* model (i.e., model where changes are not direcly pushed on the public source code, but are first reviewed by the mainatiners).

2.1.3 Source Code Diff

The most common way to show differences between two files (e.g., to study a source code patch) is to use the `diff`[3] utility, or other tools based on it (e.g., `git diff`[4]). Such utilities calculate and show line-based differences

[1]`https://github.com/`
[2]`https://git-scm.com`
[3]`https://www.gnu.org/software/diffutils/`
[4]`https://git-scm.com/docs/git-diff`

between two given files (i.e., *inserted* and *deleted* lines), based on the Hunt algorithm [27] or variants of it. The output itself is often called *diff*, too.

These tools can also be used to generate an *edit script*: a sequence of actions necessary to transform the first file into the second one. A diff, alongside the modifications, usually displays some unchanged lines, in order to provide a *context* to the patch. Every group of added or deleted lines surrounded by the corresponding context in the output is called *hunk*. Figure 3.2 contains an example of diff with four hunks; in this particular case, all the unchanged lines surrounding the hunks are displayed, while usually the context shown around each group of changes is fixed (e.g., one line above and below every hunk, like in the diff in Figure 5.5).

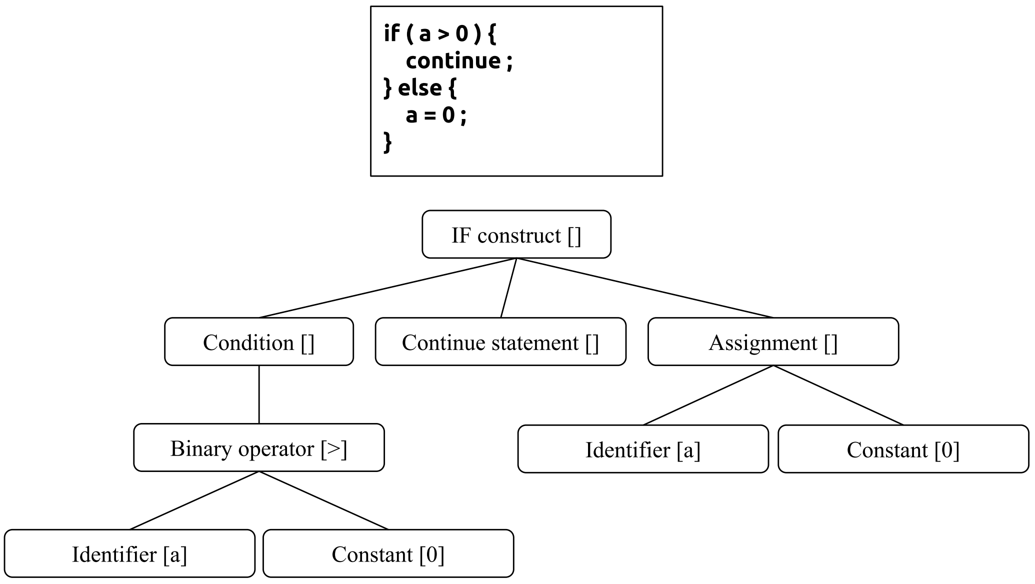
2.1.4 Source Code Analysis

Different program analysis techniques can be used to analyze computer software in order to understand its behavior in a specific situation or regarding a specific property (e.g., the presence of vulnerabilities). The analysis can be performed with or without executing the program (i.e., dynamically and statically, respectively). In this work we only refer to static analysis techniques.

2.1.4.1 Static Analysis

Static analysis is the process of analyzing programs without executing them. It could rely on simple techniques such as pattern-matching (i.e., looking for specific token patterns in the code) or on formal methods such as *symbolic execution*. Most of the static analysis techniques are performed on machine-readable representations of the source code such as the *abstract syntax tree* (AST) or the *control flow graph* (CFG). We will give an overview of these two representations, of the symbolic execution technique, and of some other concepts needed to understand the remaining parts of this work.

The CFG is a graph that represents all the paths that the program might traverse during its execution. Figure 3.1 shows an example of the CFG of a simple program. Most of the static analysis techniques rely on this representation. Usually, every CFG node contains a *basic block* (i.e., a sequence of statements that are always executed exactly once, in order, whenever the first one is executed). The CFG of a program, or function, is often used together with the corresponding *control dependency graph* (CDG): a directed



```
if ( a > 0 ) {
  continue ;
} else {
  a = 0 ;
}
```

Figure 2.1: Example of an AST with the corresponding source code. Every node contains its type followed by its value, that can be empty.

graph that represents control dependency of the statements towards each other. A statement `S1` has a control dependency on a preceding statement `S0` if the execution of `S0` determines whether `S1` is executed or not (e.g., all the statements inside a loop have a control dependency on the loop instruction). The CFG and the CDG are often used also to extract information about *dominators*; in this work we use the concept of *postdominance*: a node $n1$ is a post-dominator of a node $n0$ if all the paths that go from $n0$ to the end of the CFG (i.e., the exit node) must go through $n1$. A post-dominator of n that is not post-dominator of other post-dominators of n is called immediate post-dominator.

The AST is a tree that represents the syntactic structure of the source code. The nodes of this tree usually have a type (e.g., if-then-else construct, user-defined identifier, etc.) and a value (e.g., the value of an identifier node). Figure 2.1 show an example of AST with the corresponding code. It is called abstract because it does not represent every detail explicitly (e.g., the one in the example does not display curly brackets, and it does not explicitly show which subtree of the `IF construct` identifies the `then` or the `else`, since it can be understood by looking at the order of the nodes).

Symbolic execution is a technique used to represent the value of the variables at a specific point in the code using mathematical expressions. These expressions are sets of constraints on the variable values, collected while analyzing the program (i.e., its CFG, usually) from the entry point to the point specified (e.g., they can be collected from assignments, conditional expressions or other statements that constrain the values that a variable can assume further in the program). The inputs of the program, on which the constraints are imposed, are *symbolic values*. This technique is a possible way to study the runtime behavior of a program without actually executing it. The main limitation of this technique is the *path explosion* [17]: symbolically executing all the possible paths that a program can traverse does not scale to large software.

2.2 Problem Statement

The open-source software (OSS) model revolutionized the software industry and prior research shows that it is more secure [18, 54] than its closed-source counterpart. However, propagating changes and patches from the main repository of an open-source software to all the *related projects* (e.g., forks) is a major problem [50]. Applying these changes just as they are could have unpredictable effects because the code bases of the related projects diverged from the original one over time, and dealing with dependencies and incompatibilities between the divergent portions of the systems requires a lot of effort [51]. For example, Android depends on a Linux kernel fork, and upgrading it with patches from the kernel main repository without thoroughly testing their effects could affect some Android features.

To avoid this problem, and still be able to keep their software up to date, the maintainers of the related projects need to cherry-pick and test all the changes that they want to apply, process that requires a lot of time and effort [6, 56]: they have to manually find suitable patches, look at the code they change, understand its behavior, adapt them to their code base and check that the whole system still works as expected after their application. For these reasons changes on the main code base of a project are usually applied on the related software with a significant delay [52]: Android 7.1.1, for example, is based on Linux kernel 3.10, while the latest release of the kernel is version 4.9.8 [2].

This problem becomes critical when we consider security patches: in these cases the fixes should propagate to all the code bases as soon as possible.

Vulnerability databases such as the CVE (Common Vulnerabilities and Exposures) were born to facilitate this process: project maintainers can take them as a reference to know which security-related patches they need to apply, without having to manually find them. Despite the existence of these databases, security patches still take a lot of time to propagate to all the project forks [10,29,38,42]. In the year 2016 the Android maintainers patched 76 publicly known vulnerabilities (i.e., CVEs) from the year 2014, two from 2013 and two from 2012, which means that 80 disclosed vulnerabilities remained unpatched in the Android code base for more than one year [1]. This attracted considerable public interest; Twitter user @RatedG4E tweeted on August 2016 [4]:

"Good to see Android/Nexus devices start to pick up patches available two years ago, check 'CVE-2014- ' in August Android Sec bulletin.".

Furthermore, as we will show in this study, it is possible that the maintainers of a project misinterpret the severity of a patched bug and fail to request a corresponding entry in a vulnerability database(e.g., a CVE ID).

An ideal solution that would help the maintainers in this process of selecting and applying changes is to build a tool capable of identifying the ones that can be applied without the need for subsequent testing, because they preserve the original functionality of the software. We call them functionality-preserving patches.

2.3 State of the Art

Source code changes and patches as research topics received a lot of attention in the past decade, due to the fact that upgrading software and fixing bugs are costly and time-consuming activities that require a lot of human effort. This section covers a comprehensive portion of prior work on these topics.

2.3.1 Vulnerability finding and exploitation

Finding *unpatched code clones* is what most of the prior research on patches in the security field focused on [29,34,35]. VulPecker [35] extracts features from the vulnerability patch *diff* and uses code similarity to find clone instances of vulnerable code, ReDeBug [29] uses a syntax-based clone detection approach based on tokens, and Li et al. [34] use static and dynamic analysis techniques. In this work we also show how our tool could be used as a vulnerability finder (see Section 2.5); however, we do not look for code clones but for instances where the function affected by a patch is still equal to the unpatched version. Brumley et al. [14], instead, show how to generate exploits for a vulnerability starting from the corresponding patch.

2.3.2 Easing the patching process

Prior research has been very active in designing approaches and building tools to ease and speed the process of patching. For example, Nistor et al. [44] focus on finding performance bugs, Son et al. [55] on repairing access control bugs in web applications, Andersen er al. [8] on easing the process of patching collateral evolutions, and other studies on helping developers in applying systematic changes [39, 59]. Most of the state-of-the-art techniques that aim at easing or automating the process of patching, like the ones listed above, target only specific bug classes [40], while in this work we define and identify a set of changes that we think could be easily automated, without focusing on a specific class. Long et al. [36], in contrast with the previously mentioned studies, use machine learning to model correct code and generate generic defects fixes, but do not focus on propagating existing patches like we do in this study. Similarly to what we do in this work, Kreutzer et al. [30] use AST differencing on changes; however, they use it to extract metrics and then use these metrics to cluster the changes by similarity, and not to determine their effect on the software.

2.3.3 Software Evolution

Many studies on source code changes perform large-scale analysis on software repositories to gain insight into the dynamics of software evolution (i.e., we can group them in the so called Mining Software Repositories field [25]). Giger et al. [24] extract source code changes features and use data mining to perform software defects prediction, while Perl et al. [45] built VCCFinder, a tool that leverages code metrics and patch features (e.g., keywords in commits) to identify vulnerability-contributing changes. In this work we do not use data mining or machine learning techniques and analyze every commit without collecting external information: we mined software repositories only to extract the changes to-be-used for the evaluation (see Section 5.1).

2.3.4 Source-code Patches Analysis

Existing approaches to analyze source-code patches rely on commit-related information such as code *diff* or commit messages [9,49,57], or look for specific patterns [41]: these tools have the advantage of being fast, lightweight, scalable and suitable to be used on large code bases. However, either they only match simple patches or they analyze information that often cannot be considered reliable: commit messages are not always a good way to understand the effect of the changes [3,11,53].

Other techniques that try to understand the semantic difference introduced by a patch using static-analysis [12,13,31,32,48] and symbolic execution [15,23,37,46] suffer from scalability issues [17]. Some approaches also require the software's build environment [16], restricting their practicality and adaptability in complex software like the Linux kernel, that has many possible configurations [5].

In this work we build a fast, lightweight and scalable tool that analyzes patches trying to undersatand their effect on the behavior of the program, and that does not require information other than the source code before and after the patch (i.e., no build environment or commit messages are needed).

2.4 Goals

A tool that can identify functionality-preserving patches could be used to monitor the main repository and automatically apply this kind of patches on a target fork. Alternatively the tool could be used to build a variant of the *git rebase*[5] feature that applies only those patches mentioned above. In this paper, we argue that a significant portion of all security-related fixes fall under the category of functionality-preserving patches.

To be effective and usable on large code bases, such a system should:

- Only rely on the original and patched version of the modified source code file, without any other additional information (e.g., commit message, build environment etc.).

[5]`https://git-scm.com/docs/git-rebase`

- Be fast, lightweight and scalable.

The goal of this work is to design, implement and evaluate a static analysis technique made specifically to target source code changes and to identify patches that could be applied with minimal testing because they are most likely functionality-preserving: we call them *non-disruptive patches*. This technique should join the positive aspects of the different previous approaches and satisfy the requirements listed above.

2.5 Contributions

Specifically, these are the actual contributions of this work:

- We give the first formal definition of non-disruptive patches and design a general technique to identify them.
- We implement GitRDone, a system based on this technique, that can work taking as input only the source code of the original and the patched file.
- We evaluate GitRDone on a 39,191 commits spanning over 10 different kernels repositories, and on 191 CVE patching commits.
- We identify 8,638 non-disruptive patches and show that GitRDone could help developers in the process of selecting and testing changes and speed-up the propagation of security fixes. Some of these patches are Linux kernel security fixes for which we could not find any linked CVE and that are still unpatched in different kernel forks.

Unlike all previous work, our approach is the first that focuses on determining which of the changes made on a given software could be propagated to related projects with minimal effort, without pre-defining specific types of changes or semantic characteristics that it should target (i.e., we do not just target patches that patch a vulnerability, introduce a bug, etc.). We will release the source code of GitRDone at the time of publication.

Chapter 3

Approach

In this chapter, we first give a formal definition of the kind of source-code patches that we want to identify and design a technique to do it (Section 3.1). In Section 3.2, then, we show the details of GitRDone, the tool that we developed based on the technique mentioned before.

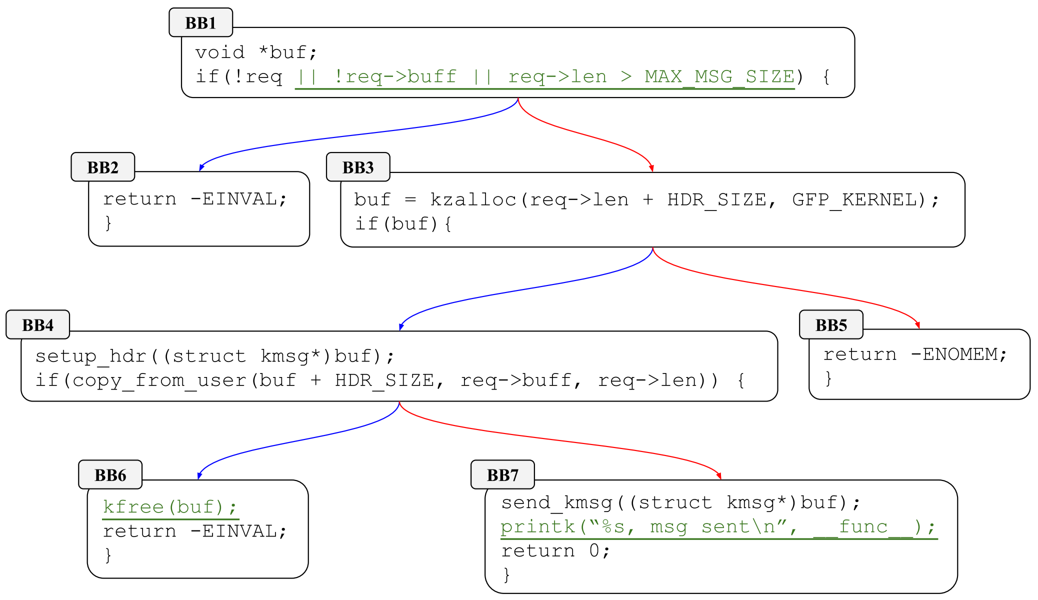
3.1 Non-Disruptive Patches

Determining that a given change preserves the original program functionality requires a full understanding of the program's dynamics, which is a task that, in the general case, can be reduced to the halting problem. However, our intuition is that if a patch satisfies certain criteria, then it can *most likely* be applied safely because it does not disrupt the original functionality of the software. We define such patches as *non-disruptive patches* (*ndp*).

Section 3.1.1 provides a detailed description of the criteria that a patch must satisfy to be identified as an *ndp*. A C language example of an *ndp* is shown in Figure 3.2, in a standard *diff* format (i.e., where + and − indicates inserted and deleted lines, respectively). Note that, while we display examples using this *diff* format, our approach interprets the changes in a different way (i.e., not only as deleted and inserted lines, see Section 3.1.1). In the example, the inserted call statements (i.e., `printk` and `kfree`) cannot disrupt the functionality, the error `return` under `!req->buff` (i.e., under the deleted condition) is still correctly handled by a preceding `if` statement, thanks to an insertion, and `req->len > MAX_MSG_SIZE` is the insertion of a previously missing length check (i.e., a fix). Figure 3.1 represents the Control flow graph

(CFG) after the application of this example patch: underlined text indicates the pieces of code inserted, while right and left children of each basic block are true and false branches, respectively.

3.1.1 Formal Definition



```
BB1:
void *buf;
if(!req || !req->buff || req->len > MAX_MSG_SIZE) {

BB2:
return -EINVAL;
}

BB3:
buf = kzalloc(req->len + HDR_SIZE, GFP_KERNEL);
if(buf){

BB4:
setup_hdr((struct kmsg*)buf);
if(copy_from_user(buf + HDR_SIZE, req->buff, req->len)) {

BB5:
return -ENOMEM;
}

BB6:
kfree(buf);
return -EINVAL;
}

BB7:
send_kmsg((struct kmsg*)buf);
printk("%s, msg sent\n", __func__);
return 0;
}
```

Figure 3.1: Control flow graph of the patched program from Figure 3.2.

First, we define some notions that will be used throughout this section:

- *State* s of a program: the snapshot of the program's data segments, including global variables, and runtime heap and stack. S indicates the set of all the possible states of a program.

- f denotes a function and any subscript to it identifies its patched version. For example: f_p indicates the function f after applying the patch p.

- *Error-handling basic blocks* (BB_{err}): basic blocks of the control-flow graph of a function that are part of its error-handling functionality. In Figure 3.1, BB2, BB5, and BB6 are error-handling basic blocks. Note that all the post-dominator basic blocks of an error-handling basic block are also error-handling basic blocks (no such blocks are present in Figure 3.1).

- *Guarding basic block (BB_{guard})*: we define the non error-handling basic block that is the immediate pre-dominator of a BB_{err} as a guarding basic block, and the corresponding condition as a *guarding condition (C_{guard})*. Depending on whether the BB_{err} is part of its true branch or false branch, the corresponding guarding condition is denoted as a positive (C^{T}_{guard}) or negative (C^{F}_{guard}) guard, respectively. In Figure 3.1, BB1 and BB4 are guarding basic blocks with positive guard, while BB3 is a guarding basic block with a negative guard.

- We use the notation $s \hookrightarrow f$ to indicate that a state s *flows* through a function f, which means that starting from the entry basic block of f with state s, none of the BB_{err}s of f can be reached. In other words, s represent a valid starting state for the function f (i.e., a state that does not result in an error).

- $\Im$ indicates a set of source code statements. $\Im_f$ and $\Im_{f_p}$ indicate the set of source code statements in f and f_P, respectively.

- $trace(s, f)$ indicates the set of source code statements of the function f that are executed when the function is run with starting state s. The following relation trivially holds for every function: $trace(s, f) \subseteq \Im_f$ (i.e., a statement, to be executed, must be present inside the executed function). Note that when we say that two traces are the same we mean that the order of the statements within them must also be the same. However, when referring to the trace as sets (e.g., checking if a statement x is present *in* the trace), the order is not relevant.

Using these notions, we want to define the concept of non-disruptive patches (*ndp*s), a class of patches that do not disrupt the original functionality of a program and that can be most likely applied with minimal testing.

We say that a statement is *modified* by p if its code is changed but the statement itself is not *moved* with respect to the other ones, not considering the ones that are *inserted* and *deleted* (i.e., one should check the relative position of the statements and not the line numbers). We assume that if a patch *modifies* the statements in a way such that the execution trace of f and f_p is the same, when they start with the same state, then it means that

```
int process_req(struct usr_req *req) {
  void *buf;
- if(!req) {
+ if(!req || !req->buff || req->len > MAX_MSG_SIZE) {
    return -EINVAL;
  }
  buf = kzalloc(req->len + HDR_SIZE, GFP_KERNEL);
  if(buf) {
    setup_hdr((struct kmsg*)buf);
-   if(!req->buff) {
-     return -EINVAL;
-   }
    if(copy_from_user(buf + HDR_SIZE, req->buff,
      req->len)) {
+     kfree(buf);
      return -EINVAL;
    }
    send_kmsg((struct kmsg*)buf);
+   printk("%s, msg sent\n", __func__);
    return 0;
  }
  return -ENOMEM;
}
```

Figure 3.2: Example of a non-disruptive patch.

the patch does not disrupt the functionality. However, we want to cover also patches that *insert*, *delete* or *move* statements.

To include deletions we relax the assumption saying that the trace should be the same, excluding the statements not present anymore after the patch: this is reasonable because if deleting a statement does not affect all the rest of the trace then its deletion is not disruptive. To consider also insertions and moves, we make another relaxation and say that the statements that are *in* the new trace should also be *in* the original trace only if they are actually present in the source code of the original function: this is reasonable because if the trace contains the same statements despite insertions, deletions and modification it means that the patch was not disruptive. To make this last relaxation more reasonable, we also enforce the fact that no new states should flow through the function, as these could make the code behave in a different way. The concepts defined above are formalized by Equation 3.1 and Equation 3.2, that define the criteria that a patch p , to a function f , needs to match in order to be *ndp*.

$$\forall s \in S \mid (s \hookrightarrow f_p) \rightarrow (s \hookrightarrow f) \qquad (3.1)$$

$$\forall s \in S.(\forall x \in trace(s, f_p) \mid (x \in \Im_f) \rightarrow (x \in trace(s, f))) \qquad (3.2)$$

Similarly, we say that a patch p is *non-disruptive* for a program if Equation 3.1 and Equation 3.2 hold for all the functions in it.

Note that this definition holds for the case of an empty patch, where $f_p = f$. We assume the patches that satisfy these criteria (i.e., *ndp*s) to be *most likely* functionality-preserving, because of the reasoning outlined above; however, patches that do not satisfy the above criteria could still be such that they preserve the original functionality. Section 3.1.2 describes the a method to identify *ndp*s as defined above.

3.1.2 Identifying *ndp*s

In Section 3.1.1 we defined *ndp*s: a class of patches that are *most likely* functionality-preserving. However, running a software and check its trace for all the possible states is not practical. In the remaining part of this section, we show a general technique to identify if a given patch is an *ndp*, that works by analyzing one by one the statements that the patch *affects* (i.e., modifies, inserts, deletes or moves). Some of the steps of this technique are implementation-specific: Section 3.2 shows how we implemented it in our system.

Consider a given patch p, with f and f_p being the targeted function before and after applying the patch, respectively. The first step of our technique is to identify all the error-handling basic blocks (BB_{err}s) in f and f_p. Then the following heuristic is applied: all the changes to the statements within BB_{err}s are discarded (i.e., not considered for the following steps). This is based on the assumption that any change to error basic blocks does not disrupt the original functionality (i.e., it just results in better or adjusted error-handling).

The remaining statements affected by p are then analyzed one by one as described below, depending on their class (e.g., conditional or non-conditional) and on the type of change that affects them. If *all* the changes to *all* these statements are identified as non-disruptive by one of the following steps then the patch p is identified as an *ndp*, otherwise it is considered disruptive.

3.1.2.1 Non-Conditional Statements

If a non-conditional statement does not perform memory writes (i.e., writes include modifications to the value of a variable), then the changes that affect it (i.e., delete, insert or modify it) are considered non-disruptive. For example, any change that affects a statement that just calls the C function `printf` is non-disruptive, unless the format string contains `%n`, which results in a write. For affected statements that perform writes, one could apply implementation-specific analyses or heuristics to determine if the effects of the changes are disruptive or not (see Section 3.2 to see how what we do in our system). For example, the insertion of certain statements that perform write operations could be considered non-disruptive (e.g., `kfree` in Figure 3.2).

3.1.2.2 Modified Conditional Statements

Consider a conditional statement $c \in \Im_f$ and the corresponding modified version $c_p \in \Im_{f_p}$: we say that the changes that p applies on the condition are non-disruptive (nd) if c and c_p satisfy Equation 3.3.

$$nd(c, c_p) = \begin{cases} (c \rightarrow c_p) & if\ c = C^T_{guard} \\ (c_p \rightarrow c) & otherwise \end{cases} \tag{3.3}$$

Equation 3.3 models the fact that if a condition is modified then it should become more restrictive in order to not allow new states to flow through the function (i.e., $c_p \rightarrow c$). However, in case the condition is a positive guard, (i.e., it guards a BB_{err}), then it should become less restrictive (i.e., $c \rightarrow c_p$). In this way the changes to the condition satisfy Equation 3.1.

3.1.2.3 Inserted Conditional Statements

Insertions of conditional statements (i.e., which means that they are only present in f_p) are considered non-disruptive (note that we are referring only to the conditional statements themselves, not to the statements in the blocks that they guard). This is because the insertion of a condition c does not make Equation 3.1 or Equation 3.2 invalid. Equation 3.2 always holds because if c is inserted then $(x \in \Im_f)$ is *false*. Equation 3.1, instead, holds because the insertion of c can only restrict the set of states that flow through the function (i.e., this happens when the condition is a C^T_{guard} and it guards an inserted BB_{err}).

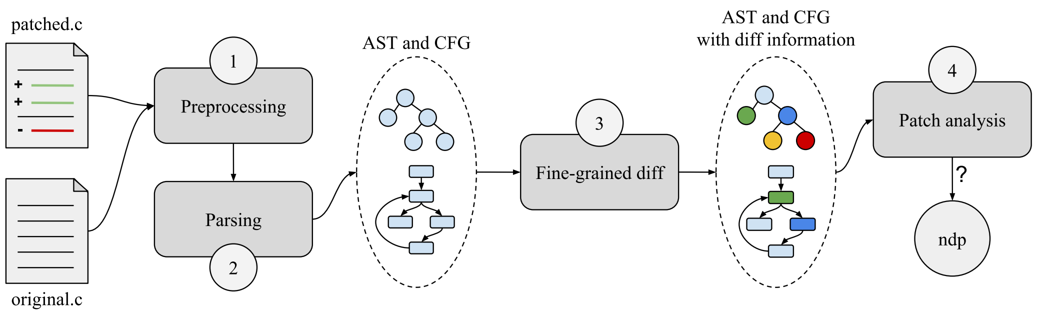


Figure 3.3: Flow of the actions performed by GitRDone.

3.1.2.4 Deleted Conditional Statements

The removal of a condition is considered a non-disruptive change if compensated by the insertion or modification of another condition. For example, in Figure 3.2 the condition `if(!req->buff)` is deleted but the first `if` is modified to handle the same cases. More formally, if the removal of a condition c does not allow more states to flow through the function, then it is non-disruptive. This can be formally expressed as shown in Equation 3.4, where BB_c is the basic block to which c belongs, $succ(BB)$ is the set of all the successors basic blocks of BB, and $s \hookrightarrow_{BB} f$ means that the execution of f with starting state s reaches the basic block BB in function f.

$$\forall\ BB \in succ(BB_c).(BB \neq BB_{err} \rightarrow (\forall s \in S \mid (s \hookrightarrow_{BB} f_p) \rightarrow (s \hookrightarrow_{BB} f))) \tag{3.4}$$

Equation 3.4 enforces the fact that if a condition is deleted, then all the basic blocks that it guards (i.e., $succ(BB_c)$) should still be reachable under the same, or under more restrictive, constraints (i.e., as enforced the by the rightmost implication in the equation).

For a patch affecting multiple functions, the above described steps should be performed for each one of them. Note that the outlined technique is just a possible method to identify *ndp*s: we acknowledge the possibility of existence of more efficient ways to achieve the same result. In Section 3.2, we show the design of our *ndp*s identification tool based on this approach.

3.2 GitRDone Design

As mentioned in Section 2.4, we want to build a tool that can identify *ndp*s just by analyzing the original and the modified versions of the patched source code file without the need for additional information (e.g., build environment, preprocessor flags, header files, commit messages etc.), and that is lightweight, fast and scalable. In this section we show the details of GitRDone, a tool that uses static analysis to analyze a given C source code patch and determine if it is an *ndp*, and that adheres to the requirements mentioned above. It is based on the *ndp*s identification approach outlined in Section 3.1.2, and we will show all the assumptions that we made to make it practical and to follow the requirements. We design our tool to work on C source files. The block diagram in Figure 3.3 shows the steps that GitRDone performs, outlined in a detailed way in the remainder of this section.

3.2.1 Preprocessing

GitRDone starts by handling the C preprocessor directives. File inclusions (i.e., `#include`) are ignored, since as a requirement we do not want to collect information outside of the two input source code files. Macro definitions are ignored too: macro calls will be treated as regular function calls as explained in further steps. The system then uses the *unifdef*[1] tool to handle conditional code inclusion directives (e.g., `#ifdef`, `#ifndef`, etc.): the output of this tool is a valid C source file, without any of these constructs. Note that this step could exclude certain code segments, depending on the constants defined within the file (i.e., the ones to which *unifdef* has access in this setup). This first step outputs two C source files ready to be parsed.

3.2.2 Parsing

The preprocessed source files are parsed using the Joern [58] fuzzy parser, which provides an Abstract Syntax Tree (AST) for all the functions in the file. Although Joern provides also a Control Flow Graph (CFG), with nodes linked to the ones in the AST, and information about uses and definitions, we had to modify it to suite to our needs. Specifically, we had to implement identification of basic blocks boundaries (since the Joern CFG nodes are

[1]`http://dotat.at/prog/unifdef/`

single statements), linking of basic blocks to the corresponding AST nodes, additional analyses (e.g., reaching definitions analysis [43]) and a simple type inference [47]. At the end of this phase GitRDone has access to AST, CFG and analysis results for each function.

3.2.3 Fine-Grained Diff

GitRDone uses function names to pair the functions in the original file with the corresponding ones in the new files, assuming patches that insert, delete or rename one or more functions to not be *ndp*. It then identifies the functions affected by the patch using *java-diff-utils*[2], a common text *diff* tool, between the two input files. Text diffing tools, based the Hunt algorithm [27], are good for understanding statements inserted and deleted by a change, but they are not useful for understanding moved statements and fine-grained differences between original and new code in terms of AST nodes. For this reason, our system then applies a state-of-the-art AST diffing technique, Gumtree [21], between the original and patched ASTs of the affected functions. GumTree maps the nodes in the old AST with the corresponding nodes in the new one and identifies nodes that have been moved, inserted, deleted or updated. A moved node is a node that the patch moved in another position in the AST, but whose content was unchanged, while an updated node is a non-moved node whose content was changed. The differences in the AST are also associated to the corresponding nodes in CFG, thanks to the links between AST and CFG nodes provided by Joern. Figure 3.4 shows an example of why we need fine-grained diff information: it is an *ndp* that just moved a function call (i.e., `pmic_spmi_show_revid(regmap, &sdev->dev);`) under a new `if` statement, without changing it, but the text diff simply displays this change as the result of two independent actions (i.e., a deletion and an insertion); GumTree, instead, maps together the corresponding nodes in the two ASTs and marks them as *moved*.

3.2.4 Patch Analysis

In the remaining part of this section we explain how GitRDone performs in practice the *ndp*s identification, based on the general technique described

[2]`code.google.com/archive/p/java-diff-utils/`

```
  if (IS_ERR(regmap))
    return PTR_ERR(regmap);

- pmic_spmi_show_revid(regmap, &sdev->dev);
+ /* Only the first slave id for a PMIC contains this information */
+ if (sdev->usid % 2 == 0)
+   pmic_spmi_show_revid(regmap, &sdev->dev);

  return of_platform_populate(root, NULL, NULL, &sdev->dev);
```

Figure 3.4: An *ndp* identified by our tool on the main Linux kernel repository (commit 742dcd115cb523f). It shows a case where the information provided by GumTree is needed to understand the semantics of the change.

in Section 3.1.2. The tool performs the following steps for every function affected by the patch.

3.2.4.1 Identification of Error Basic Blocks

A basic block BB is marked as a BB_{err} when it ends with a `return` statement that returns a constant value or a C standard error code (i.e., one of the constant symbols defined in `errno.h`) prepended by a minus sign (e.g., $BB2$, $BB5$, and $BB6$ in Figure 3.1). Cases that do not fall into this category (e.g., BBs that end with calls to an error-handling function or with a variable-value `return` statement) are hard to handle, without complex inter-procedural analysis or symbolic execution to keep track of variable values and constraints. GitRDone uses the following heuristic to handle these other cases in a lightweight fashion: it checks for the presence of error-related words (e.g., `panic, error, fatal`) or C error standard codes in the AST of the last two statements of the basic block, meaning that it can find them both in identifiers (i.e., variables, function names, etc.) and in constant strings; we defined a set of 15 of these error-related words, based on our observation. This heuristic is based on the insight that error-handling basic blocks usually log or print a message and then execute an error-related statement (e.g., `return error_var;`, `goto fatal_state;`, an error-handling function call, etc.). In addition, also `break` and `continue` are considered error-related words (since we observed that they are mostly used to handle some loop-related error states, such as loop early interruptions to improve performance) and also `return` statements with no value. Note that this heuristic based approach could lead to false positives and that we expect every error basic block to be singularly matched by this method, thus the post-dominators of a BB_{err} are not automatically considered BB_{err}s themselves. GitRDone then discards all changes that happen within the identified BB_{err}s (i.e., it does not consider them for the following steps).

3.2.4.2 Non-Conditional Statements

As explained in Section 3.1.2, GitRDone needs to determine the effects of every affected non-conditional statement on the state of the function. This cannot be done in practice, in the general case. However, our system applies the following heuristic: the changes that affect statements that cannot control the flow of the patched function are considered non-disruptive. We say that a statement cannot control the flow of a function if its variable definitions (i.e., *defs*, variables to which it assigns a value) are such that they cannot reach any corresponding *use* inside any conditional statement (affected or not). For example, in the sequence `V = 0; V = 1; if(V) return;` the first statement cannot control the flow because, although `V` is used in an `if` condition, the value that it assigns to `V` cannot reach the use of `V` in such condition. The second assignment, however, can control the flow (i.e., 1 reaches the `if` condition). This process is applied recursively: for example, in the sequence `V = 0; V1 = V; if(V1) return;` the first definition can reach the use of `V1` in the `if` condition. This is based on the insight that if a change cannot affect the flow of the execution in any way, then it is *most-likely* non-disruptive. Note that this heuristic is applied only on the statements present in the patched version of the function (i.e., it is not applied on the ones deleted by the patch, addressed later in this section).

As an additional heuristic for this step, GitRDone ignores the *defs* performed by the following classes of statements:

- Identifier declarations that do not perform an initialization assignment (i.e., `int a;`), or where the initialization assignments matches one of these other heuristics.
- Memory erasure and release calls (e.g., `memset(buff, 0, sizeof(buff);`, `kfree(buff);`).
- Zero-value or NULL-value assignments (e.g., `ptr = NULL;`, `i = 0;`).

- Size-setting assignments matched by looking for the words `size` and `sizeof` on the left and right side, respectively, (e.g., `size = sizeof(buff);`) and error assignments, matched by looking for the same set of words used in the identification of error basic blocks (e.g., `err_code = -EINVAL`).
- Assignments where all the identifiers in the AST of the right expression respect the C constants standard uppercase convention (e.g., `mask = PLLMB_MISC1_LOCK_ENABLE`).

The insight behind this heuristic is that all the effects of these classes of statements are *most likely* related with resetting some variable values or memory locations, or with setting up values for further operations (e.g., returning an error, saving the size of a buffer to enable further checks, etc.). GitRDone also ignores changes that affect label statements (including their insertion and deletion), since they do not have any effect: what changes the behavior of the program are changes in the corresponding `goto` statements, that are studied as any other non-conditional statement.

For what concerns *deleted* non-conditional statements, GitRDone relies on the following assumption: if the effects of all the changes performed by a the analyzed patch, excluding statements deletions, are identified as non-disruptive, then the deletions of non-conditional statements are ignored. The reason behind this assumption is that we consider ignoring these statements, in case all the rest is non-disruptive, as an heuristic to practically apply what we say at the beginning of the formal definition (Section 3.1.1), when we allow a patch to also delete some statements. In addition, if a patch only deletes statements, then it is not considered an *ndp*.

3.2.4.3 Inserted Conditional Statements

Following the general *ndp*s identification technique outlined in Section 3.1.2, GitRDone considers all the insertions of conditional statements (i.e., `if-else` statements in C source code) to be non-disruptive. The statements in the code blocks that they guard, if affected, are handled singularly by the other steps of the analysis. Statements that are just moved under an inserted condition, without any modification, are ignored (i.e., not considered as affected by all the other steps, including the non-conditional statements one). For example, the function call in Figure 3.4 (i.e., `pmic_spmi_show_revid(regmap, &sdev->dev);`). This because their functionality is not changed: the new condition just restricts the number of states that can reach them.

3.2.4.4 Modified Conditional Statements

GitRDone uses the Z3 theorem prover [19] to prove the implication between the original and the patched conditions as described in Section 3.1.2. Every affected condition is converted into a Z3 formula using information from the AST and creating a new Z3 unconstrained variable for every operand. Furthermore, the tool constraints the unsigned variables to be greater than zero and defines standard C type limits as Z3 constants (i.e., the ones defined in `limits.h`). Type casts and kernel optimization macros (e.g., `likely()` and `unlikely()`) are ignored. In the current implementation bitwise operations are disabled by default and not handled; they can be enabled but the result is a huge performance overhead (i.e., up to 50 times the current execution time, for some patches). This means that every bitwise operation is converted into a Z3 unconstrained variable, just as a normal operand (e.g., when proving $(v1\&v2 > v3 + 1) \rightarrow (v1\&v2 > v3)$, GitRDone actually proves that $Z3var1 > Z3var2 + 1 \rightarrow Z3var1 > Z3var2$ using Z3). Note that this step is applied not only on `if/else` conditions but also on loop conditions.

3.2.4.5 Deleted Conditional Statements

For what concerns *deleted* conditional statements, in order to implement what Section 3.1.2 says in the corresponding step, we should use symbolic execution and Z3 to implement a path-sensitive analysis that collects all the constraints and proves the resulting implication (see Equation 3.4). This would result in possible scalability issues, because of the path explosion [17] problem. To keep the tool the most lightweight possible, we apply the same heuristic used in the analysis of deleted non-conditional statements, described above, and ignore deletions of the conditional ones too: the reason for which we think that doing this is reasonable is also the same (refer to the end of the non-conditional statements analysis outlined in this section).

3.2.5 Additional Heuristics

To increase the amount of patches that GitRDone can handle, we consider as non-disruptive the ones where *all* the changes fall into *a single one* of the following categories:

- Insertion or deletion of kernel synchronization function calls (e.g., `spin_lock`, `spin_unlock`, `mutex_lock`, `mutex_unlock`).

- Modification of C security-sensitive function calls (e.g., `strcpy`, `strncpy`, `strlcpy`, `memcpy`, `sprintf`, `sscanf` and their variants).
- Off-by-one fixes (e.g., `size_str = sizeof(buf) -1` when `-1` is inserted or deleted).

The fact that *all* the changes must fall into *a single one* of these categories means that for one these heuristics to hold, for example, no conditions should be affected, as they do not fall in any of these groups. The reason behind these heuristics is that all the changes that they match are recurring patterns of fixes that are *most likely* non-disruptive. Furthermore, we consider a patch that *only* deletes statements to be an *ndp* when these are all inside error basic blocks.

3.3 Alternative Ideas and Solutions

What we described up to this point is the final outcome of months of challenging work: we started from slightly different goals and from the application of different ideas and solutions that did not always work as expected, and eventually ended up trying with the approach that we outlined above. In this section we talk about how the goals of the work evolved over time and we give an overview of the other ideas and solutions that we tried to apply. Note that, however, the motivation of the research never changed over time.

The initial goal of the work was, given a source-code patch, to identify the set of security-related changes within it and to automatically apply them only if non-disruptive with respect to the original functionality. The insight behind this idea is that there could be patches that include unrelated changes (i.e., tangled code changes [26]), with only a portion of them fixing a security issue and the rest doing something different that could disrupt the software functionality. At this stage of the project we still did not have a proper definition of non-disruptive patch: the idea was to consider as non-disruptive the read-only changes (i.e., changes that do not modify variable values or memory locations).

The initial approach that we designed was the following:

1. Extract the source-code diff of the patch.
2. Extract, for every hunk in the diff, a set containing the variables that it reads and writes.

3. Untangle unrelated code changes by grouping together the diff hunks for which these sets of variables intersect.

4. Determine which groups of hunks perform security-related changes.

5. If a group of hunks is security-related and read-only, apply its changes.

In order to be able to analyze the hunks we linked all the inserted and deleted statements to the corresponding AST nodes and, for each hunk, extracted the fine-grained AST diff, using the same technique previously described in this chapter. Eventually, we figured out that we were trying to achieve too many goals together: 1) untangling code changes; 2) identifying security-related changes; 3) identifying non-disruptive changes; 4) automatically apply non-disruptive security-related changes. For this reason, we decided to start by focusing on a single one of these goals: identifying if a source-code patch is security-related or not.

In order to achieve this new goal, we first started by manually analyzing and studying a set of CVE-patching commits with the aim of identifying common traits and features that characterize security patches. We observed that a good portion of them simply modify or insert conditional statements to restrict the possible inputs that can enter into one or more basic blocks. Furthermore, we noticed that most of these patches modify the flow of the program in a way that makes more paths than before enter into error basic blocks.

For this reason, we started implementing the identification of error basic blocks and the analysis of the conditional statements in a way similar to the one described previously in this chapter. Eventually, while trying to apply this first analysis and studying the results, we realized that we were actually identifying patches that did not disrupt the original functionality, but that many of them were not security fixes. This is because all the security fixes that we observed in order to model the approach were also non-disruptive, and while looking for common traits we ended up modeling this characteristic.

For this reason, we realized that just the identification of non-disruptive patches represents a complex and open research problem, and decided to focus on it, since we already had a good idea of what to look for. We then decided to define in a more formal way what we meant by non-disruptive and the result was what we described in this chapter.

Chapter 4

Implementation Details

GitRDone has been implemented as a Java-based tool. We chose to use this programming language because we found Joern and GumTree to be exactly what we needed when looking for a fuzzy parser and a tool capable of computing AST diffs; being these two components programmed in Java, the easiest way to extend them or interact with them is using the same language. The remaining part of this section outlines the implementation details of the steps described in Section 3.2.

4.1 Input handling and Preprocessing

GitRDone, once built, is a `.jar` file that needs to be called providing two files as input. It then reads the content of the files and performs the pre-processing step using `unifdef`, as described in Section 3.2.1. In particular, it calls the `unifdefall` executable on the two files separately and saves the output in two new files that will be the ones parsed and analyzed in the further steps. GitRDone also accepts different parameters as input: the relevant ones are `-BV` and `-vis`. The first one can be used to enable bitwise operations when analyzing conditional statements (see Section 3.2.4) and the second one to enable the graphical visualization of the AST difference (see Section 4.3.1).

4.2 Parsing and Joern Extensions

4.2.1 Functions AST Extraction

Joern is a tool written in Java and built to be flexible and easily extensible. GitRDone sets up the C fuzzy parser by instantiating a `ModuleParser` object: it calls the constructor `ModuleParser(ANTLRParserDriver driver)` with an `ANTLRCModuleParserDriver` object as parameter. To be able to extract the AST for the functions in the files we implemented our own extensions of the `ASTWalker` and `ASTNodeVisitor` classes; an instance of our AST walker is set as observer of the `ModuleParser` by calling the method `parser.addObserver`. In this way, for every node parsed, the parser calls the `processItem` method of the walker; the walker, then, calls our visitor that visits the node and saves it in a list if it is an instance of `FuncionDef` (i.e., the root node of a function in the AST). GitRDone then gets the list of references to all the function ASTs from the visitor, for both the original and the new file.

4.2.2 CFG and Basic Blocks

To extract the CFG of a function using Joern it is sufficient to instantiate an `ASTToCFGConverter` and to call its `convert(FunctionDef f)` method passing as parameter a function AST (i.e., one of the `FunctionDef` objects extracted as described above). We had to implement the CFG traversal functionality: whenever traversing a CFG, GitRDone goes through all the nodes once, ignoring loops. The Joern CFG contains a statement in every node, with a reference to the corresponding node in the AST, but it does not contain basic blocks information; in order to identify the basic blocks we implemented a simple algorithm that works in two steps: 1) Identification of block leaders; 2) Identification of BBs.

To perform the leaders identification GitRDone traverses the CFG and performs the following checks for every node n:

- If n is the first statement it is a leader.
- If n has more than one outgoing edge, the destination nodes of these edges are leaders: this is because n is a conditional statement and all the nodes that follow it start new basic blocks.

- If n has more than one outgoing edge, the immediate post-dominator of n is a leader: this is because n is a conditional statement, and the next node that it is executed independently from the condition in n starts a new basic block.
- If n is a C label statement it is a leader.

The information about post-dominators can be extracted directly using Joern: it is sufficient to instantiate a `DominatorTree<CFGNode>` object and call its `getDominator(CFGNode n)` method. To instantiate it we need to create a CDG (Control Dependency Graph) by executing `CDGCreator cdgCreator = new CDGCreator(); CDG cdg = cdgCreator.create(cfg);`. We then call `cdg.getDominatorTree()`.

To perform the *BB*s identification starting from the leaders we then proceed as follows: given a leader, the corresponding *BB* starts with it and contains all the statements that follow it and that precede the following leader. However, the Joern CFG is not always sound and does not handle properly the CFG edges that exit from some statements for which it has no information about the semantic (e.g., `exit()` should go directly to the end, but since Joern does not enter the function's code, being a fuzzy parser, it just considers it as any other function call and adds it as a common statement node). For this reason we added an heuristic to consider some kind of statements as basic block closers: `break`, `continue`, and `return` statements, and calls to `exit()` or to `longjmp()`. Furthermore, we also added the following heuristic: if after the *BB*s extraction, performed as described, some groups of consecutive statements do not belong to any *BB*, then they are marked as a new *BB* (this holds also for single statements).

4.2.3 Reaching Definitions

We needed to implement the recursive lookup of reaching definitions [43] to perform the analysis of non-conditional statements as described in Section 3.2.4. Joern is already able to extract uses and defs for a given statement, and provides use-def relations through a DDG (Data Dependency Graph). A use-def relation links to the use of a variable, performed by a specific statement, the definition (or definitions) of that variable that can reach it *non recursively*, with reference to the defining statement. For example, consider the code `A = 0; B = A; return B;`. When getting the reaching definitions for the use of `B` performed by the `return` statement, Joern returns only `B = A`, while we want to have also information about the fact that `A = 0` can influence, indirectly, the returned value. To implement the full reaching definition analysis we had to add the recursive lookup of the definitions that can reach a given use; the heuristic used in the analysis of non-conditional statements (see Section 3.2.4) can then be implemented as a method that returns all the statements that have at least one def that can recursively reach at least one use performed by a given statement, passed as argument. The pseudocode of this function can be found in Figure 4.1.

To implement it using Joern we needed to create:

- A `UDG` (Use-Def Graph), by calling the method `convert(CFG cfg)` of the `CFGToUDGConverter` module.
- A `DefUseCFG`, by calling the method `convert(CFG cfg, UDG udg)` of the `CFGAndUDGToDefUseCFG` module.
- A `DDG`, by calling the method `createForDefUseCFG(DefUseCFG ducfg)` of the `DDGCreator()` module.

The `DefUseCFG` can then be used to extract uses and defs for a given statement (by calling its `getSymbolsDefinedBy` and `getSymbolsUsedBy` methods) and the `DDG` to extract the non-recursive reaching definitions (by calling its `getDefUseEdges` method and looking for the proper relations in the set of `DefUseRelation` objects that it returns).

4.2.4 Type Inference

We needed to implement a simple type inference [47] in order to be able to understand which variables are unsigned, so that they can be constrained while using `Z3` (as described in Section 3.2.4). The type inference is performed in two steps: 1) extraction of known types for variables, by looking at the declarations; 2) inference of the types of the other variables by looking at the assignments. The first step simply looks at variable declarations to identify a set of known variable-type pairs to start with, while the second step goes through all the assignments in the code, recursively, and propagates the type of the variable on the right side of an assignment to the variable on the left side; recursively means that this step is repeated until a fixed point is reached (i.e., no more types can be inferred). As an additional heuristic, also type

```
def get_reaching_defs_recursive(stmt) {
  return get_reaching_defs_recursive_helper(stmt, [], [])
}

def get_reaching_defs_recursive_helper(stmt, analyzed_stmts, retval) {
  if stmt in analyzed_stmts
    return
  else
    analyzed_stmts.add(stmt)

  uses = get_vars_used_by(stmt);

  for use in uses {
    // this gets the defining statements
    reaching_defs = ddg.get_non_recursive_reaching_defs(stmt, use);
    retval.add_all(reaching_defs);

    for definition in reaching_defs
      get_reaching_defs_recursive_helper(definition, analyzed_stmts, retval)
  }
}
```

Figure 4.1: Pseudocode of the recursive extraction of statements whose defs can reach uses of another given statement.

casts are used to infer the type of the variable on the left (i.e., when there is an explicit cast, the type is known). Casts and identifier declarations can be simply found by traversing the AST and looking at the corresponding node types.

4.3 Functions Diff

After parsing the two files, extracting the functions ASTs and pairing them by name, as described in Section 3.2.2 and Section 4.2, the first step to perform the diff is understanding which of the functions are affected by the patch. In Section 3.2.2 we said that we use *java-diff-utils* to do it: specifically, we use it to extract the line-based diff between the two files and then look at the line numbers. Joern provides line information (i.e., line in the source-code file) for every AST node, so it is easy to understand which functions are affected by the patch. For all these affected functions, we then convert the original and new AST in the format that GumTree accepts (i.e., `ITree`) and extract the tree diff information.

The conversion of the Joern AST into an `ITree` is trivial: we just create

```
Matcher m = Matchers.getInstance().getMatcher(itree1, itree2);
m.match();
MappingStore mappings = m.getMappings();
ActionGenerator g = new ActionGenerator(itree1, itree2, mappings);
g.generate();
List<Action> actions = g.getActions();
```

Figure 4.2: Java code to extract the GumTree mappings and actions, given two `ITree`s.

```
-void foo(int a, int b){
-  printf("a: %d, b: %d", a, b);
+void foo(int a){
+  printf("a: %d", a);
 }
```

Figure 4.3: Example code corresponding to the ASTs shown in Figure 4.5 and Figure 4.6.

a new `ITree` node for every node in the AST, fill it with the same data (i.e., node type and label), and link them together in the same way. GumTree mappings and diff can then be generated by calling the code in Figure 4.2; GumTree returns a list of ordered actions that, if applied, transform the original AST into the new one (i.e., an edit script).

To map these actions as diff information on the Joern AST, GitRDone just goes through the list and maps every node in the original and new AST to an action type (i.e., delete, insert, move, update), if affected by an action. Note that it is necessary to use both the ASTs because information about deleted nodes can be added only to the original one, since such nodes are not present in the new one, and, for the same reason, information about inserted nodes can be added only to the new one. Furthermore, move and update actions are always added to the original and new nodes involved: the Gumtree mapping can then be used to understand, for example, where a node in the original AST was moved. For this reason, we also keep a mapping between nodes of a Joern AST and the corresponding tree in the GumTree-compatible format, so that these GumTree mappings between original and new nodes can be easily accessed.

```
 void foo(int a){
-   if (a < 0)
+   if (a < 0 || a >= 10)
      return -1;
 }
```

Figure 4.4: Example code corresponding to the ASTs shown in Figure 4.7 and Figure 4.8.

4.3.1 Visualization

GitRDone can output a graphical representation of the original and new ASTs; it also includes diff information. Figure 4.5 and Figure 4.6 show the visualization of the original and new ASTs, respectively, corresponding to the source code diff shown in Figure 4.3. Another example is shown in Figure 4.7 and Figure 4.8, with the corresponding code shown in Figure 4.4. We used Graphviz[1] to implement the visualization feature: GitRDone converts the AST in the `.dot` format and uses the `dot` command to generate the corresponding visualization in the desired format (e.g., `.png`, `.svg`, etc.).

4.4 Patch Analysis

GitRDone uses the information extracted as shown previously in this chapter to perform the analysis described in Section 3.2.4. Specifically, it uses the CFG and the AST, together with the basic blocks information, the type inference and the reaching definitions, to apply the described analysis of the statements. The only part of the implementation of this portion of the tool that is worth showing in a more detailed way is how conditions are converted in order to be analyzed by Z3.

The Z3 interaction is performed through the corresponding Java API: for every condition GitRDone goes through the corresponding original and new ASTs and generates the necessary operations and variables. The two conditions are converted into the same Z3 context so that the tool is aware of the fact that a defined variable with a certain name is the same in both. Eventually, to prove the implication between the two conditions, GitRDone tries to verify that the negation of the implication is unsatisfiable. For example, to check if it is always true that $OriginalC \rightarrow NewC$ using Z3, one

[1]`http://www.graphviz.org/`

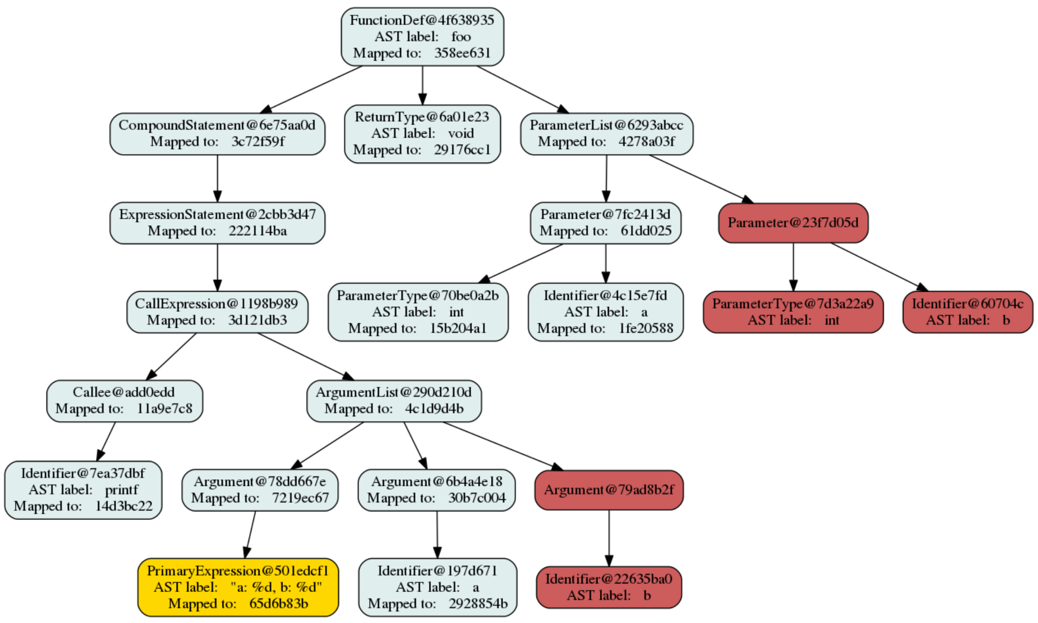

FunctionDef@4f638935 AST label: foo Mapped to: 358ee631
CompoundStatement@6e75aa0d Mapped to: 3c72f59f
ReturnType@6a01e23 AST label: void Mapped to: 29176cc1
ParameterList@6293abcc Mapped to: 4278a03f
ExpressionStatement@2cbb3d47 Mapped to: 222114ba
Parameter@7fc2413d Mapped to: 61dd025
Parameter@23f7d05d
CallExpression@1198b989 Mapped to: 3d121db3
ParameterType@70be0a2b AST label: int Mapped to: 15b204a1
Identifier@4c15e7fd AST label: a Mapped to: 1fe20588
ParameterType@7d3a22a9 AST label: int
Identifier@60704c AST label: b
Callee@add0edd Mapped to: 11a9e7c8
ArgumentList@290d210d Mapped to: 4c1d9d4b
Identifier@7ea37dbf AST label: printf Mapped to: 14d3bc22
Argument@78dd667e Mapped to: 7219ec67
Argument@6b4a4e18 Mapped to: 30b7c004
Argument@79ad8b2f
PrimaryExpression@501edcf1 AST label: "a: %d, b: %d" Mapped to: 65d6b83b
Identifier@197d671 AST label: a Mapped to: 2928854b
Identifier@22635ba0 AST label: b

Figure 4.5: Example of original AST with diff information; the corresponding source code diff is shown in Figure 4.3, the new AST in Figure 4.6.

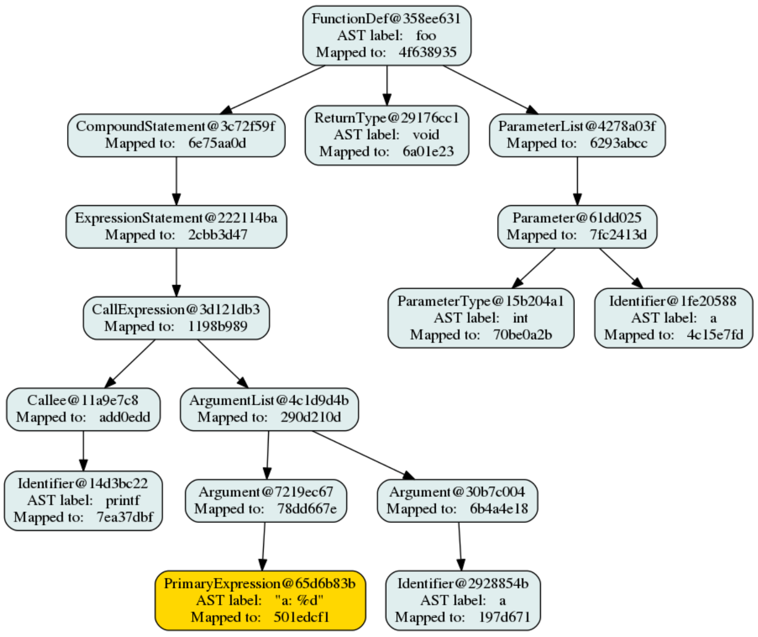

FunctionDef@358ee631 AST label: foo Mapped to: 4f638935
CompoundStatement@3c72f59f Mapped to: 6e75aa0d
ReturnType@29176cc1 AST label: void Mapped to: 6a01e23
ParameterList@4278a03f Mapped to: 6293abcc
ExpressionStatement@222114ba Mapped to: 2cbb3d47
Parameter@61dd025 Mapped to: 7fc2413d
CallExpression@3d121db3 Mapped to: 1198b989
ParameterType@15b204a1 AST label: int Mapped to: 70be0a2b
Identifier@1fe20588 AST label: a Mapped to: 4c15e7fd
Callee@11a9e7c8 Mapped to: add0edd
ArgumentList@4c1d9d4b Mapped to: 290d210d
Identifier@14d3bc22 AST label: printf Mapped to: 7ea37dbf
Argument@7219ec67 Mapped to: 78dd667e
Argument@30b7c004 Mapped to: 6b4a4e18
PrimaryExpression@65d6b83b AST label: "a: %d" Mapped to: 501edcf1
Identifier@2928854b AST label: a Mapped to: 197d671

Figure 4.6: Example of new AST with diff information; the corresponding source code diff is shown in Figure 4.3, the original AST in Figure 4.5.

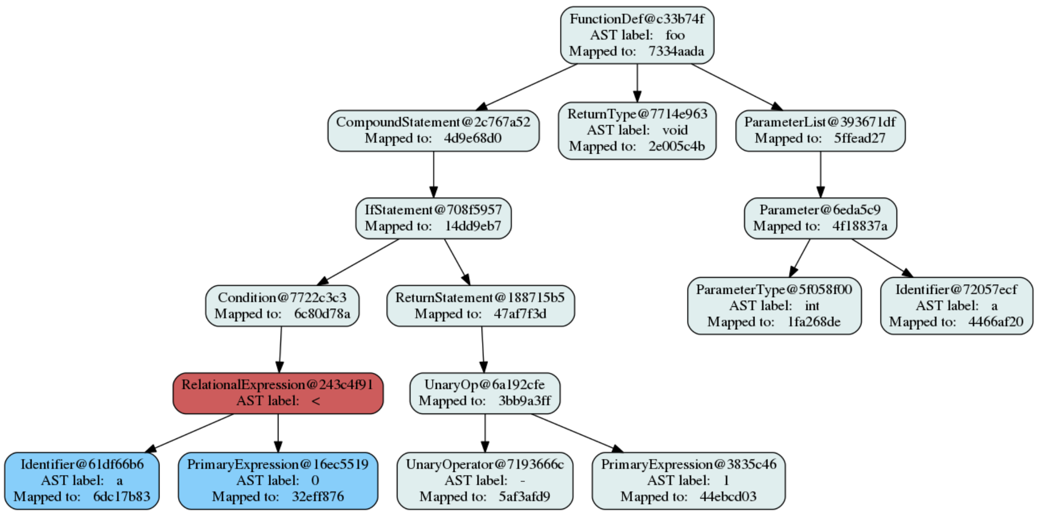


Figure 4.7: Example of original AST with diff information; the corresponding source code diff is shown in Figure 4.4, the new AST in Figure 4.8.

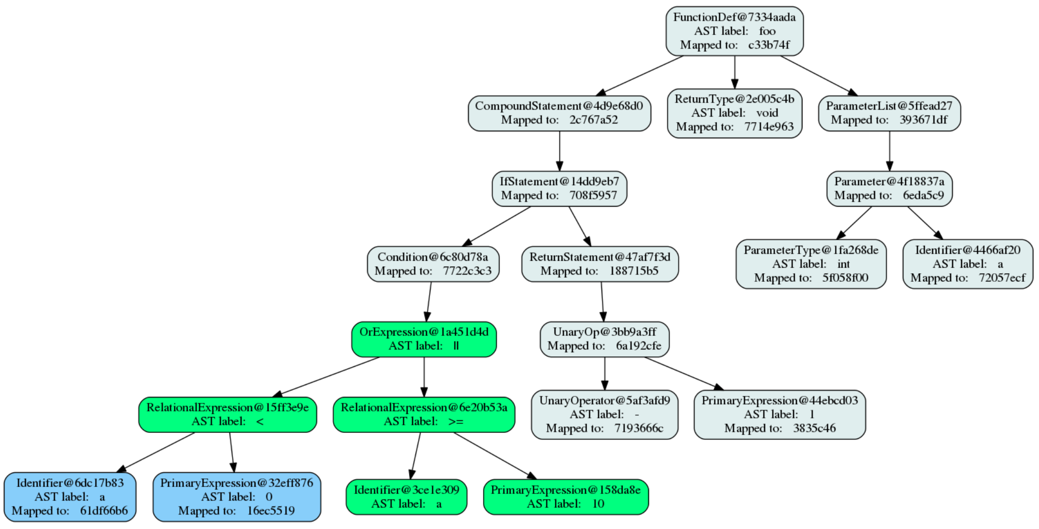


Figure 4.8: Example of new AST with diff information; the corresponding source code diff is shown in Figure 4.4, the original AST in Figure 4.7.

must check the satisfiability of $\neg(OriginalC \rightarrow NewC)$: if it is unsatisfiable (i.e., no values are such that the formula holds), then the first implication is always true.

Chapter 5

Evaluation

We evaluate the effectiveness of GitRDone in three different ways. First, we run it on a large dataset of 39,191 changes (i.e., commits) spanning over 10 kernels repositories, collected over the year 2016, in order to understand if it actually detects *ndp*s, according to our definition (see Section 5.1). Second, we run it on a set of security patches (i.e., CVE patching commits) to evaluate the usefulness of this tool in speeding the propagation of these more critical fixes. Third, we check if some of the *ndp*s identified on the Linux kernel are still unpatched, at the time of writing, on some of its Android-related forks: this would provide real examples where GitRDone can be useful and, in case of security fixes, would show a way to use it as a vulnerability finding tool (see Section 5.3). We argue that the results of these three experiments will demonstrate the usefulness that such a tool can represent for project maintainers, when it comes to ease and speed the process of applying software patches.

The analysis that GitRDone performs, described in Section 3.2, is an intra-functional static analysis that does not consider the interaction between the different modified functions. For this reason, to isolate the effect of these interactions, that represent a possible confounding factor, we evaluate it only on patches that affect a single C source file (i.e., `.c` format only) and a single function within it. We also exclude the patches where the changes are completely trimmed by the preprocessing step (see Section 3.2). All the patches studied in these experiments are real changes extracted from repositories of widely used kernels (see Section 5.1 for more details).

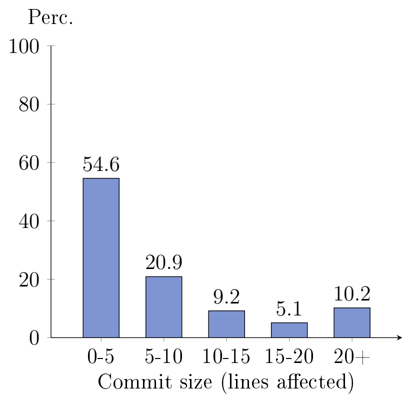


Figure 5.1: Distribution of the size of all the commits studied by GitRDone.

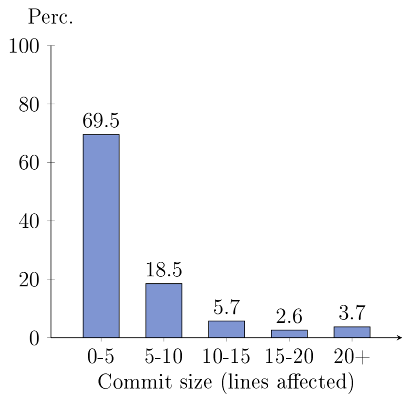


Figure 5.2: Distribution of the size of the *ndp*s identified by GitRDone.

5.1 Experiment 1: Large-Scale Evaluation

In this experiment we run GitRDone on a large set of patches: we selected ten open-source kernels widely used by desktop, mobile and embedded operating systems, and we collected from each of them all the single-C-file single-function commits of the year 2016 (considering merges as single commits). The targeted repositories, together with information about the number of commits studied, are shown in Table 5.1.

Table 5.3 shows the number of *ndp*s identified by GitRDone in each one of the 10 studied projects, together with some statistics that show the number of cases where the patch performs only condition-related non-disruptive changes, as we model them in Section 3.2. (e.g., it just modifies an existing

	Project	studied git branch/tag	studied commits
1	Linux kernel main repository	master	19,393
2	NVIDIA Tegra Linux kernel	android-7.0.0_r0.3	259
3	Qualcomm Msm Linux kernel	android-7.1.0_r0.3	1,857
4	OP-TEE Trusted OS	master	58
5	LK embedded kernel main repository	master	16
6	QuIC LK codeaurora embedded kernel	LE.UM.1.1.10-00410-8x09.0	186
7	Xiaomi Linux kernel	land-m-oss	417
8	Xperia Linux kernel	aosp/LA.BR.1.3.3_rb2.14	463
9	Linaro ARM Linux kernel	optee	16,472
10	Android x86_64 Linux kernel	android-7.0.0_r0.32	69
Total			39,191

Table 5.1: Studied projects and commits. (refer to Table 5.2 for project names).

Project	URL
1	`https://github.com/torvalds/linux`
2	`https://android.googlesource.com/kernel/tegra`
3	`https://android.googlesource.com/kernel/msm`
4	`https://github.com/OP-TEE/optee_os`
5	`https://github.com/littlekernel/lk`
6	`https://source.codeaurora.org/quic/la/kernel/lk`
7	`https://github.com/MiCode/Xiaomi_Kernel_OpenSource`
8	`https://github.com/sonyxperiadev/kernel`
9	`https://github.com/linaro-swg/linux`
10	`https://android.googlesource.com/kernel/x86_64.git/`

Table 5.2: URLs of the studied repositories (refer to Table 5.1 for project names).

condition). The column that counts the patches that just inserted new `if` statements takes into account also the ones where they were added together with the basic blocks that they guard (whose affected statements must still pass the *ndp* checks that GitRDone performs).

We show these statistics because we think that the fact that so many patches just modify existing conditions or insert missing checks is an interesting finding: these are cases where the maintainers would most likely just need to check very few lines of code, that often do not even perform writes (i.e., assignments within conditions are not common), in order to choose if to apply the patch or not. Furthermore, these numbers show that GitRDone does not detect just these short patches.

```
                error = -EINVAL;
                goto out_put_tmp_file;
        }

+       if (f.file->f_op != &xfs_file_operations ||
+           tmp.file->f_op != &xfs_file_operations) {
+               error = -EINVAL;
+               goto out_put_tmp_file;
+       }
+
        ip = XFS_I(file_inode(f.file));
        tip = XFS_I(file_inode(tmp.file));
```

Figure 5.3: A security patch identified as *ndp* by GitRDone on the main Linux kernel repository (commit 3e0a3965464505). *It does not have a corresponding CVE ID.*

		patch just affects specific statements (% over *ndp*s)	
Project	***ndp*s (% over commits)**	**just inserts if stmts**	**just affects conditions**
1	4,239 (21.86%)	764 (18.02%)	412 (9.72%)
2	72 (27.80%)	25 (34.72%)	9 (12.50%)
3	474 (25.51%)	104 (21.94%)	67 (14.14%)
4	17 (29.31%)	6 (35.29%)	3 (17.65%)
5	4 (25.00%)	1 (25.00%)	0 (0.00%)
6	36 (19.35%)	18 (50.00%)	1 (2.78%)
7	92 (22.06%)	24 (26.09%)	15 (16.30%)
8	117 (25.27%)	31 (26.50%)	12 (10.26%)
9	3,567 (21.65%)	660 (18.50%)	353 (9.90%)
10	20 (28.99%)	5 (25.00%)	1 (5.00%)
Total	8,638 (22.04%)		

Table 5.3: Large-scale experiment results (refer to Table 5.1 for project names and information).

	sample size	**true**	**false**
identified as *ndp*s (positive matches)	100	96	4
not identified as *ndp*s (negative matches)	100	86	14

Table 5.4: Results of the manual checking performed on the commits analyzed in the first experiment (see Table 5.3).

		patch just affects specific statements	
CVE patches source	***ndp*s**	**just inserts if stmts**	**just affects conditions**
Linux	36 / 69 (52.17%)	12 (33.33%)	5 (13.89%)
Android bulletin	73 / 122 (59.84%)	22 (30.14%)	15 (20.55%)
Total	109 / 191 (57.06%)		

Table 5.5: CVE patches experiment results.

True positive rate	96.00 %
False positive rate	4.00 %
True negative rate	86.00 %
False negative rate	14.00 %
Precision	96.00 %
Recall	87.27 %

Table 5.6: Statistics of the manual checking results shown in Table 5.4.

```
  u32 count, ordinal;
  unsigned long stop;

+ if (bufsiz < TPM_HEADER_SIZE)
+   return -EINVAL;
+
  if (bufsiz > TPM_BUFSIZE)
    bufsiz = TPM_BUFSIZE;
```

Figure 5.4: A security patch identified as *ndp* by GitRDone on the main Linux kernel repository (commit ebfd7532e98581). *It does not have a corresponding CVE ID.*

Over the total 39,191 commits studied, GitRDone identified 8,638 non-disruptive patches. Given the large amount of subjects, we manually checked 100 patches identified as *ndp*s and 100 not identified as *ndp*s, randomly-sampled from all the studied projects. The results of the manual-checking are shown in Table 5.4, while Table 5.6 shows the corresponding statistics. The false positives are due to the fact that the technique described in Section 3.1.2 and the assumptions and heuristics outlined in Section 3.2 are not sound. However, since GitRDone does not apply the patches in an automatic way, the false positives have no direct negative impact, assuming that the maintainers are experienced enough to understand that they differ from the rest of the patches that GitRDone suggests to apply.

Figure 3.4 shows a real instance of *ndp* on the Linux kernel, identified by GitRDone. Although it is a simple and easily readable example, we argue that the tool works also on larger patches: this can be inferred by looking at Figure 5.2 and Figure 5.1. The first of these two plots (i.e., Figure 5.2) shows the distribution of the size of the commits that GitRDone identified as *ndp*, while the second one (i.e., Figure 5.1) shows the same information for all the ones that it analyzed. One can see that the 6.3% of the patches identified as *ndp*s (i.e., 544 of them) affect more than 15 lines of code (i.e., lines affected are the sum of lines added and deleted according to the *git diff* tool), even though they are on average smaller than the average studied patch. Figure 5.5 shows an example of a more complex patch identified by GitRDone.

Looking at these results we feel confident in saying that GitRDone would be helpful for project maintainers. It could directly be used, for example, to prioritize the changes that must be analyzed and tested, so that the ones that need less testing can be applied first. In this way, according to the results shown in Table 5.6, the 96% of the patches that it identifies as *ndp*s could be imported in a fast and almost effortless way, leaving as last all the ones that are not *ndp*s (i.e., those patches for which it would be time consuming anyway).

In Section 2.2 we mentioned the fact that it is possible for some patched vulnerabilities to not have a corresponding entry in the CVE database, because of human error (we selected the CVE database because it is the one used as a reference by the maintainers of the studied projects). While manually checking the commits studied in this experiment we confirmed that this problem exists: we found real instances of security patches that GitRDone marked as *ndp* and for which we were not able to find a corresponding entry in the CVE database. Figure 5.4 and Figure 5.3 show two of these instances.

5.2 Experiment 2: Evaluation on CVEs

The goal of this second experiment to determine if it is true that a significant portion of security patches are *ndp*s, as claimed in Section 2.2. We collected all the CVE patching commits linked as reference fixes for kernels CVEs from all the 2016 Android security bulletins [1], and, as previously done in the first experiment, we studied only the ones that patch a single function and a single C file. This resulted in the analysis of 122 CVE patches over the total 238 C-modules patches extracted from the bulletins (the rest of them patch CVEs in `C++` or Java modules).

In order to have a larger dataset we also scraped the CVE database, extracted the Linux kernel CVEs and collected the commit hash of every patch linked in the references of each one of them. We then looked for these CVE-patching commits in the dataset of the first experiment (see Section 5.1). Eventually we were able to find a total of 69 CVEs, in addition to the ones collected from the Android bulletins. Table 5.5 shows the results obtained

```
@@ -787,3 +787,3 @@ int kvm_vcpu_ioctl_config_tlb(struct kvm_vcpu *vcpu,
  if (ret < 0)
-   goto err_pages;
+   goto free_pages;

@@ -792,3 +792,3 @@ int kvm_vcpu_ioctl_config_tlb(struct kvm_vcpu *vcpu,
    ret = -EFAULT;
-   goto err_put_page;
+   goto put_pages;
  }
@@ -798,3 +798,3 @@ int kvm_vcpu_ioctl_config_tlb(struct kvm_vcpu *vcpu,
    ret = -ENOMEM;
-   goto err_put_page;
+   goto put_pages;
  }
@@ -803,8 +803,12 @@ int kvm_vcpu_ioctl_config_tlb(struct kvm_vcpu *vcpu,
         GFP_KERNEL);
+ if (!privs[0]) {
+   ret = -ENOMEM;
+   goto put_pages;
+ }
+
  privs[1] = kzalloc(sizeof(struct tlbe_priv) * params.tlb_sizes[1],
         GFP_KERNEL);
-
- if (!privs[0] || !privs[1]) {
+ if (!privs[1]) {
    ret = -ENOMEM;
-   goto err_privs;
+   goto free_privs_first;
  }
@@ -815,3 +819,3 @@ int kvm_vcpu_ioctl_config_tlb(struct kvm_vcpu *vcpu,
    ret = -ENOMEM;
-   goto err_privs;
+   goto free_privs_second;
  }
@@ -847,12 +851,10 @@ int kvm_vcpu_ioctl_config_tlb(struct kvm_vcpu *vcpu,
  return 0;
-
-err_privs:
- kfree(privs[0]);
+ free_privs_second:
  kfree(privs[1]);
-
-err_put_page:
+ free_privs_first:
+ kfree(privs[0]);
+ put_pages:
  for (i = 0; i < num_pages; i++)
    put_page(pages[i]);
-
-err_pages:
+ free_pages:
  kfree(pages);
```

Figure 5.5: An *ndp* identified by our tool on the main Linux kernel repository (commit 46d4e7479252d3).

```
 int check_aboot_addr_range_overlap(uint32_t start, uint32_t size)
 {
        /* Check for boundary conditions. */
-       if ((start + size) < start)
+       if ((UINT_MAX - start) < size)
                return -1;
```

Figure 5.6: Real integer overflow patch identified as *ndp* by GitRDone (CVE-2014-9795 from July 2016 Android security bulletin).

```
        alts = &iface->altsetting[fp->altset_idx];
        altsd = get_iface_desc(alts);
+       if (altsd->bNumEndpoints < 1) {
+               kfree(fp);
+               kfree(rate_table);
+               return -EINVAL;
+       }
+
        fp->protocol = altsd->bInterfaceProtocol;

        if (fp->datainterval == 0)
```

Figure 5.7: Real CVE patch identified as *ndp* by GitRDone (CVE-2016-2184 patched by Linux kernel commit 0f886ca12765d2).

after running GitRDone on them, together with the same kind of statistics shown in the results of the first experiment (i.e., Table 5.3).

The results of this second experiment show that the 57.06% of the CVE-patching commits are non-disruptive, while in the first experiment GitRDone (i.e., on generic patches) the percentage was 22.04%. These findings demonstrate that for a security patch it is *most likely* to be non-disruptive then for a generic patch. They also show that GitRDone could be useful not only to speed-up the process of selecting and applying a significant number of changes (as shown in Section 5.1) but also to apply the 57.06% of the security patches in a faster way and to notify maintainers about some of the patches that fix a security flaw but do not have an associated CVE database entry (i.e., as shown in Section 5.1, see Figure 5.4 and Figure 5.3). These security fixes would otherwise fall into the pool of all the commits that project maintainers need to manually analyze, without a way to distinguish them from the others.

Figure 5.6 and Figure 5.7 shows two example of CVE patching commits identified as *ndp*s by GitRDone, extracted from the Android security bulletin and the Linux kernel, respectively. Figure 5.6, in particular, shows one of the CVEs that we mentioned in Section 2.2: it was patched in Android with more than one year delay from the appearance of the corresponding entry in the database.

5.3 Experiment 3: Zero-Days in Vendor Kernels

In this third experiment we check how many of the Linux Kernel mainline commits identified as *ndp*s in the first experiment (see Section 5.1) still have to be applied to one or more of the Linux Kernel forks that we studied (i.e., projects 2, 3, 7, 8, 9, and 10 in Table 5.1), at the time of writing (i.e., first week of February 2017). To do that, given a commit identified as *ndp*, we extract the affected file's source code before the change and we compare it to the same file, if present, in all the listed kernel forks (Table 5.1 show the git branch or tag that we studied). If the function affected by the commit does not differ between these two files then it means that the change has not been applied on the studied fork. To perform the comparison we use the *git diff* tool and check that there are no modifications in the targeted function (i.e., *git diff* is able to determine the C functions changed).

After applying this technique we found that 880 of the 4,239 Linux kernel identified *ndp*s (i.e., 20.75%) are still not applied in at least one of the considered forks: we identified 2076 instances of this behavior, meaning that most of these 880 commits are still unapplied on more than one fork. A significant portion of these are changes not considered useful by the maintainers (e.g., removals of unused code, small refactorings, etc.), and not imported for this reason. However, we found out that 18 of them are CVE patching commits (i.e., the ones that we linked to the corresponding CVEs, as shown in Section 5.2) that still have to be imported by the maintainers of some forks: this supports the findings of previous studies [10,29,38,42] that say, as already mentioned in Section 2.2, that vulnerability databases are not always effective in speeding the propagation of security fixes.

While manually looking for critical patches in this set of commits we also found 5 instances of security patches that do not have a corresponding CVE database entry and that are still unapplied on different kernel forks, including even the ARM Linux kernel main repository (i.e., project 9 in

Table 5.1): these can be seen as potential zero-day vulnerabilities. We will report all of them to the corresponding project maintainers and vendors and submit all the necessary requests for CVEs. For security reasons, we do not disclose more information about these patches.

5.4 Performance Considerations

We measured the time that GitRDone takes to run on 20,000 of the patches studied in the first experiment (i.e., Section 5.1) and we can argue that it can be considered fast and scalable: the average time that it takes to analyze a patch is 1.14 seconds when running on a machine equipped with two 2.40 GHz 6-core 12-thread CPUs and 100GB of RAM.

5.5 Summary of Results

Based on the results outlined throughout this section, we can argue that our tool can be helpful for project maintainers: the propagation of patches from the main repository of a project to the related software is, in general, slow, and requires a lot of effort, as shown in Section 2.2, and this is true also when it comes to security patches. The maintainers have to manually monitor a specific source in any case: either the commit log of the main line, when they want to perform generic upgrades, or some vulnerability database (i.e., CVEs, in the studied projects), when they want to apply the latest critical fixes. GitRDone would instead identify a portion of commits (i.e., the *ndp*s) that are most likely applicable with minimal testing effort, and would directly notify the maintainers about them. Furthermore, these *ndp*s, as we shown in the results, include also a significant portion of the CVE-patching commits (i.e., 57.06%) and instances of security fixes that patch vulnerabilities not present on the CVE database, meaning that GitRDone can be useful in patching vulnerabilities that would, otherwise, remain unpatched for a long time. In addition, we also shown how GitRDone could be used to find zero-day vulnerabilities in related projects starting from the software on which they depend on.

Chapter 6

Conclusions

In this work we provided a formal definition of *non-disruptive* patches (*ndp*s) and outlined a method to identify them: to the best of our knowledge, this is the first study that does that. We also designed, implemented and evaluated GitRDone, a tool based on our *ndp* identification approach that can determine if a patch is non-disruptive using only the original and the patched source code of the affected file, without the need for external information (e.g., build environment, commit message, etc.). Our large scale evaluation on 39,191 commits extracted from 10 different open source kernels repositories, and on 191 CVE patches, shows that we identify *ndp*s with a good precision (i.e., 96.00%) and that a significant amount of security patches are non-disruptive (i.e., 57.06%). Furthermore, we show how GitRDone can be used to find various unpatched security issues (i.e., zero-day vulnerabilities) in the studied vendor kernels repositories, starting from security *ndp*s applied on the mainline repository and not linked to any CVE. We are in the process of reporting the discovered vulnerabilities to the affected vendors.

6.1 Limitations

This study comes with several limitations. In Section 3.1.1, we defined *ndp*s but we did not provide a formal proof that shows that an *ndp* does not actually disrupt the original functionality of the software. In Section 3.1.2 we then defined a general technique to identify them: as shown by our evaluation, even a simplified implementation of this technique leads to good results in practice. We acknowledged the possibility of false positives, but our experiments show that the they are a small percentage in practice, and that they do not represent a significant risk for the maintainers.

In the current implementation, GitRDone performs an intra-functional analysis (see Section 3.2) and studies every modified function independently, reason for which in we evaluated it on patches that affect just a single file and a single function (see Chapter 5). The system can be extended to multiple functions, task that involves handling the inter-functional interactions [22, 28]. We acknowledge that there could be *ndp*s that affect multiple functions and files; the evaluation of GitRDone on these is out of the scope of this study. Furthermore, previous research show that most of the commits are small [7] in repositories where developers are experienced, such as the studied ones (see Table 5.1). We also expect kernels developers, given their experience, to not commit tangled changes [26], but the tool could also be extended to be capable of untangling code changes before the analysis [20].

Another limitation is that GitRDone, at the moment, works only on C source code; however, the parser that we use should be easily extensible to other languages. The fine-grained diff step is language agnostic, thus, to extend the tool to other languages, we would only need to add language specific heuristics and preprocessing. A good solution would be to have a configurable front end for different languages (i.e., approach already used by LLVM [33]). As our implementation is based on Joern and GumTree, we also share the same limitations that these tools have.

6.2 Future Work

While GitRDone can be used to identify non-disruptive patches, we think it represent only a first step towards a system that can also automatically apply them and that could potentially be integrated in version control systems (e.g., *git*). Then, in order to be automatically applicable on different code bases, the patches should be abstracted in a context-aware fashion and transformed to suite to a different context (e.g., with different identifiers names), using an approach such as the one shown by Meng et al. [39]. Even in the current state, we plan to enable the integration of GitRDone with a version control system, so that it can monitor the commit log of a project and notify the maintainers of related repositories (i.e., fork of such project) about possible *ndp*s. This would already speed the patch propagation process and help patching more security issues (because of those security *ndp*s that miss a corresponding CVE, as mentioned above). To enable future research we will release the source code of GitRDone at the time of publication.

Bibliography

[1] 2016 android security bulletins. `source.android.com/security/bulletin/2016.html`. Accessed: 2017-02-11.

[2] Android msm kernel. `https://android.googlesource.com/kernel/msm.git/+/android-msm-angler-3.10-marshmallow-mr1`. Accessed: 2017-02-13.

[3] The biggest and weirdest commits in linux kernel git history. `www.destroyallsoftware.com/blog/2017/the-biggest-and-weirdest-commits-in-linux-kernel-git-history`. Accessed: 2017-02-15.

[4] Community reaction to delayed patching. `https://twitter.com/RatedG4E/status/760322614912954368`. Accessed: 2017-02-13.

[5] Linux kernel configuration. `http://www.tldp.org/HOWTO/SCSI-2.4-HOWTO/kconfig.html`. Accessed: 2017-02-13.

[6] John Admanski and Steve Howard. Autotest-testing the untestable. In *Proceedings of the Linux Symposium*. Citeseer, 2009.

[7] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I Maletic. What's a typical commit? a characterization of open source software repositories. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 182–191. IEEE, 2008.

[8] Jesper Andersen, Anh Cuong Nguyen, David Lo, Julia L Lawall, and Siau-Cheng Khoo. Semantic patch inference. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 382–385. IEEE, 2012.

[9] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement?: A text-based approach to classify change requests. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08, pages 23:304–23:318, New York, NY, USA, 2008. ACM.

[10] Ashish Arora, Ramayya Krishnan, Rahul Telang, and Yubao Yang. An empirical analysis of software vendors' patch release behavior: impact of vulnerability disclosure. *Information Systems Research*, 21(1):115–132, 2010.

[11] Gabriele Bavota. Mining unstructured data in software repositories: Current and future trends. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 5, pages 1–12. IEEE, 2016.

[12] David Binkley. Using semantic differencing to reduce the cost of regression testing. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 41–50. IEEE, 1992.

[13] David Binkley, Rob Capellini, L Ross Raszewski, and Christopher Smith. An implementation of and experiment with semantic differencing. In *Software Maintenance, 2001. Proceedings. IEEE International Conference on*, pages 82–91. IEEE, 2001.

[14] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 143–157. IEEE, 2008.

[15] Raymond P.L. Buse and Westley R. Weimer. Automatically documenting program changes. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 33–42, New York, NY, USA, 2010. ACM.

[16] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Pro-*

ceedings of the 33rd International Conference on Software Engineering, pages 1066–1071. ACM, 2011.

[17] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[18] Russell Clarke, David Dorwin, and Rob Nash. Is open source software more secure? *Homeland Security/Cyber Security*, 2009.

[19] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[20] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. Untangling fine-grained code changes. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 341–350. IEEE, 2015.

[21] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324. ACM, 2014.

[22] Jaroslav Fowkes, Razvan Ranca, Miltiadis Allamanis, Mirella Lapata, and Charles Sutton. Autofolding for source code summarization. *arXiv preprint arXiv:1403.4503*, 2014.

[23] Debin Gao, Michael K Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In *International Conference on Information and Communications Security*, pages 238–255. Springer, 2008.

[24] Emanuel Giger, Martin Pinzger, and Harald C Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 83–92. ACM, 2011.

[25] Ahmed E Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48–57. IEEE, 2008.

[26] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 121–130. IEEE, 2013.

[27] James Wayne Hunt and MD MacIlroy. *An algorithm for differential file comparison*. Bell Laboratories New Jersey, 1976.

[28] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, volume 1, pages 2073–2083.

[29] Jiyong Jang, Abeer Agrawal, and David Brumley. Redebug: finding unpatched code clones in entire os distributions. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 48–62. IEEE, 2012.

[30] Patrick Kreutzer, Georg Dotzler, Matthias Ring, Bjoern M Eskofier, and Michael Philippsen. Automatic clustering of code changes. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 61–72. ACM, 2016.

[31] Shuvendu K Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *International Conference on Computer Aided Verification*, pages 712–717. Springer, 2012.

[32] Shuvendu K. Lahiri, Kapil Vaswani, and C A. R. Hoare. Differential static analysis: Opportunities, applications, and challenges. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 201–204, New York, NY, USA, 2010. ACM.

[33] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.

[34] Hongzhe Li, Hyuckmin Kwon, Jonghoon Kwon, and Heejo Lee. A scalable approach for vulnerability discovery based on security patches. In *International Conference on Applications and Techniques in Information Security*, pages 109–122. Springer, 2014.

[35] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: an automated vulnerability detection system based on code

similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 201–213. ACM, 2016.

[36] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms and search spaces for automatic patch generation systems. 2016.

[37] Paul Dan Marinescu and Cristian Cadar. Katch: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 235–245. ACM, 2013.

[38] Miles A McQueen, Trevor A McQueen, Wayne F Boyer, and May R Chaffin. Empirical estimates and observations of 0day vulnerabilities. In *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*, pages 1–12. IEEE, 2009.

[39] Na Meng, Miryung Kim, and Kathryn S McKinley. Systematic editing: generating program transformations from an example. *ACM SIGPLAN Notices*, 46(6):329–342, 2011.

[40] Martin Monperrus. Automatic software repair: a bibliography. *University of Lille, Tech. Rep. hal-01206501*, 2015.

[41] Alessandro Murgia, Giulio Concas, Michele Marchesi, and Roberto Tonelli. A machine learning approach for text categorization of fixing-issue commits on cvs. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 6. ACM, 2010.

[42] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 692–708. IEEE, 2015.

[43] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis.* Springer, 2015.

[44] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. C aramel: detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 902–912. IEEE Press, 2015.

[45] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 426–437. ACM, 2015.

[46] Suzette Person, Matthew B Dwyer, Sebastian Elbaum, and Corina S Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 226–237. ACM, 2008.

[47] Benjamin C Pierce and David N Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

[48] Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, and Vinay Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 188–197. IEEE, 2004.

[49] Sarah Rastkar and Gail C. Murphy. Why did this code change? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1193–1196, Piscataway, NJ, USA, 2013. IEEE Press.

[50] Baishakhi Ray and Miryung Kim. A case study of cross-system porting in forked projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 53:1–53:11, New York, NY, USA, 2012. ACM.

[51] Eric Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999.

[52] Eric Rescorla. Security holes... who cares? In *USENIX Security*. Washington, DC, 2003.

[53] Eddie Antonio Santos and Abram Hindle. Judging a commit by its cover: correlating commit message entropy with build status on travis-ci. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 504–507. ACM, 2016.

[54] Guido Schryen. Security of open source and closed source software: An empirical comparison of published vulnerabilities. 2009.

[55] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. Fix me up: Repairing access-control bugs in web applications. In *NDSS*, 2013.

[56] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes?: An exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 51:1–51:11, New York, NY, USA, 2012. ACM.

[57] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. Relink: Recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 15–25, New York, NY, USA, 2011. ACM.

[58] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 590–604. IEEE, 2014.

[59] Tianyi Zhang, Myoungkyu Song, Joseph Pinedo, and Miryung Kim. Interactive code review for systematic changes. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 111–122. IEEE Press, 2015.