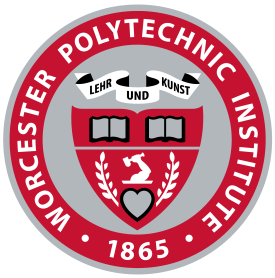

WPI

Software Safety Assurance for the Boot Power Management Processor on NVIDIA's Tegra SoC

A Major Qualifying Project submitted to the faculty
of the Worcester Polytechnic Institute
in partial fulfillment of the requirements for the
Degree of Bachelor of Science

Goutham Deva
Robert Harrison
Tejas Rao

Submitted to Professor Mark Claypool, Department of Computer Science, WPI
Sponsored by NVIDIA Corporation

March 1, 2019

Abstract

NVIDIA's Tegra System-on-a-Chip (SoC) contains a small coprocessor responsible for initializing every other component of the chip, called the Boot Power Management Processor (BPMP). NVIDIA intends to deploy the Tegra SoC inside autonomous vehicles, but in order to do this, the code running on each of its components (beginning with the BPMP) needs to be heavily refactored in order to comply with international safety standards. Our project is to assist the Tegra System Software team with these changes by writing unit tests to achieve complete code coverage, refactoring individual modules to ensure source code compliance with the MISRA C:2012 standard, and writing documentation on each functional module in the system. We performed each of these tasks successfully, contributing over five thousand lines of changes and writing one full set of architecture and design documents. These changes ought to reduce the probability of a fault in the software that would lead to a fatal error.

Contents

List of Figures

Acknowledgments

We would like to thank the following individuals and organizations in particular for their support and assistance in completing this project:

- Matt Longnecker and Sivaram Nair, for their guidance on the tools, resources, and processes we needed to use in order to assist them in ensuring the safety of the BPMP firmware.
- NVIDIA Corporation, for providing us with a project with a direct impact on the work they do, and supporting us with staff and management throughout.
- Mark Claypool, for his work organizing the project, and his dedication to advising it during its course.

Chapter 1

Introduction

NVIDIA is a technology company known for its development of the GeForce graphics processor series and the Tegra System-on-a-Chip (SoC), a high-performance mobile processor used in a diverse range of products, such as the NVIDIA Shield, a television streaming box (Daniel, 2018); the Nintendo Switch gaming console (Karandikar, 2017); and NVIDIA Drive, a simulation framework for autonomous vehicles (Smith & Ho, 2015). Every Tegra SoC contains an internal co-processor, the Boot Power Management Processor (BPMP), which is responsible for bootstrapping the system once it is powered, generating clock signals and delivering appropriate power to the main Tegra processing units (NVIDIA, 2015). Regardless of whether the Tegra SoC is deployed in a streaming box, game console, or autonomous vehicle, the BPMP executes the same firmware. To facilitate development of these disparate devices, BPMP firmware code is maintained separately from other Tegra firmware code.

In 2018, NVIDIA sought to deploy the Tegra SoC inside cars with the intent of using them to pilot autonomous vehicles. However, international standards such as ISO 26262 (ISO, 2018) demand that NVIDIA follow a rigorous set of self-defined standards before their software may be deployed inside a commercially produced vehicle. In

order to comply with these standards, NVIDIA requires that the BPMP firmware must satisfy the latest guidelines of the Motor Industry Software Reliability Association (MISRA)'s C standard, published in 2012 (MISRA, 2013). NVIDIA also requires that the BPMP codebase contain requirements-based unit tests that cover all lines of code and all branches of execution that the code runs.

Before the project began, the BPMP firmware was based on a Linux kernel for ARM processors called `littlekernel`, or `lk`, which is suitable for many embedded applications, but is neither MISRA-clean nor unit-tested to the necessary standards. It was estimated that the BPMP firmware code contained between 11,900 and 15,800 MISRA violations in total, not including those in `lk` (static analysis tools are incapable of fully listing MISRA violations). Additionally, only 14.6% of lines and branches in the codebase were covered by unit tests. In order to meet the required safety standards, the MISRA violations must be resolved, unit test coverage must be raised to 100%, and `lk` must be replaced with a safety-certified, MISRA-clean kernel.[1] Team leaders estimate that these tasks will not be completed before April 2020, well after the recommended deadline of January 2020.

The goal of our project is to assist the Tegra System Software team with achieving full compliance, with compliance meant to minimize the likelihood of a fault in the code causing software failure in production. In particular, we wrote unit tests, refactored code to achieve compliance with MISRA's standard, wrote documentation on the contents of the codebase and the processes used to create it, and restructured the architecture of the BPMP firmware to better align with the functional requirements of the system.

The team wrote software design documents for libraries used by the BPMP firmware,

[1]The BPMP firmware team elected to use a real-time operating system called SafeRTOS, developed by Wittenstein High Integrity Systems, who advertises that SafeRTOS has been externally verified to the ISO 26262 standard. This decision and the associated work falls outside the bounds of our project.

which describe the ideal operation of the software system and its functional requirements within the larger codebase. The team also wrote unit tests to increase line and branch coverage of files in the BPMP firmware to meet requirements of software safety guidelines. At the same time, the team cleaned files in the BPMP codebase of MISRA violations. Finally, the team extracted functionality from BPMP drivers related to a debugging file system and moved it to a separate location. This was done so that the debugging code could be examined separately using NVIDIA's code visualization tools, as they do not need to be under the same safety restrictions as the drivers themselves.

The rest of this report is organized as follows. The second chapter provides background on the Tegra SoC and the BPMP inside it and describes the development environment, safety standards, and code standards that NVIDIA uses on this project. The third chapter discusses what the team did over the course of the project. The chapter is divided into three distinct sections, one for design, implementation, and verification. Each high-level task falls under one of these categories. The fourth chapter lists the results of our work and its impact on the BPMP firmware team. The final chapter concludes what our team has done and discusses future steps for ensuring the safety of NVIDIA's BPMP firmware.

Chapter 2

Background

The first section of this chapter provides background on the Tegra SoC and the BPMP, as well the environment that NVIDIA uses to develop software for Tegra. The next section discusses the safety standards and requirements that NVIDIA is updating their codebase to conform to, beginning with an overview of the automotive safety standards ISO 26262 and the Automotive SPICE process reference model. Finally, we describe how these standards are applied at NVIDIA as well as the code standards that the project leads have decided to use. A discussion of the engineering verification and validation model (V-model), which describes a development process consisting of three interconnected steps, is also included.

2.1 Tegra

The term “Tegra” refers to a System-on-a-Chip designed by NVIDIA for use in various embedded projects, including the NVIDIA Shield set-top box (Daniel, 2018), the Tesla Model S (Shapiro, 2016), and the Nintendo Switch gaming console (Karandikar, 2017). The Tegra X2, the latest iteration of the SoC, contains six ARMv8 processor cores, two

of which feature a custom architecture, and a 256-core NVIDIA graphics processor (Franklin, 2017). Tegra is targeted towards high-performance, low-power environments as it boasts one teraflop[1] of computing power and consumes only seven and a half watts of power while doing so.

The BPMP is an ARM-based coprocessor inside the Tegra SoC's physical package. When the Tegra is powered on, the BPMP is the first component to receive power and is responsible for bringing up every other processor in the SoC. Additionally, the BPMP handles power management, clock management, and reset controls for the entire SoC instead of the main CPUs (NVIDIA, 2017). The BPMP communicates with the main CPUs using a shared "mailbox" interface, which consists of a dedicated memory area and a series of interrupt-enabled hardware lines that serve as "doorbells". When that interrupt fires, the receiving processor, which could be the BPMP or any main CPU, will access the shared memory and read the message there.

2.2 Development Environments

NVIDIA's system software team performs most of their work on desktop workstations running Ubuntu 16.04. Ubuntu gives users access to the `apt` package manager, which provides a large repository of packages of free and non-free software for the Linux platform. As a long-term support release, Ubuntu 16.04 will be supported by Canonical until 2021 (Ubuntu, n.d.).

NVIDIA uses Git, the distributed version control system, to manage their projects. All version control systems offer users the ability to commit code and push it to a central repository, but Git's main advantage over centralized version control systems is that it allows users to create branches, which allow authors to maintain multiple

[1] one trillion 32-bit floating-point operations per second

independent versions of a code repository that diverge from the established codebase. Users can switch between these branches with a single, short command. To resolve differences between two branches, Git allows users to merge two branches, which creates a new commit on the target branch that incorporates all of the changes from the source branch. Git also allows users to perform an interactive rebase operation, which is used to change the first divergent commit (the "base") of a branch. This feature can also be used to rewrite the history of a branch to remove or combine work-in-progress and other extraneous commits before pushing it to a central server. The two methods are often used together: if branch B is rebased on top of branch A, then branch B's commit history is the entire commit history of branch A, followed by all of the changes made on branch B. This makes merging B into A simple, since they would now share the same commit history.

NVIDIA manages a central Git server for all of its teams which runs an application called Gerrit Code Review, which was originally developed by Google's Android team for another version control system called Subversion, which does not have a branching scheme like Git's (Gerrit Code Review, n.d.). Gerrit forces software developers to get each commit approved before it is integrated into the codebase at large (usually as part of the `master` branch), which requires developers to develop directly on that branch. Despite the fact that this process appears to be inconsistent with Git's own branching model, it is possible to perform development using branches by interactively rebasing the source branch so it only consists of one commit, then submitting it for review; if the commit is approved, it is applied to the head of the target automatically.

2.3 Safety Standards and Requirements

For the Tegra SoC to be used in autonomous vehicles, NVIDIA must ensure that the hardware and software system adheres to the guidelines and rules put in place by ISO 26262. To do so, NVIDIA is retroactively creating documentation for their design process that aligns with the guidelines set in place by the Automotive SPICE Process Reference Model. We describe these guidelines below.

2.3.1 ISO 26262

ISO 26262 is a group of safety procedure guidelines to be followed when installing electrical systems on vehicles over 3500 kilograms (ISO, 2018). This standard is enforced throughout the BPMP firmware to ensure its safety. Only the first, fourth, and sixth sections of the standard were relevant to our work.

The first section of ISO 26262 establishes common terminology used throughout the standard to describe functional safety requirements for product development. This section defines fault, failure, and error as distinct defects in a system. In particular, "a fault can manifest itself as an error ... and the error can ultimately cause a failure" (ISO, 2018). The term "Safety Case" refers to a clear, comprehensive and defensible argument that the system is acceptably safe to operate in a particular context. The use of this specific terminology increases the readability and conciseness of the BPMP firmware's functional requirements.

The fourth section of ISO 26262 focuses on identifying functional safety requirements for each phase of system development. These procedures are meant to help define system properties and determine the behavior of each component of the system. Examples include developing technical safety requirements, defining system constraints, as well as designing integrated tests within the project. These func-

tional safety requirements are derived from business requirements for the system at large, and also consider how the system handles any failure of these components to do what they are required to.

The sixth section of ISO 26262 provides procedures for product development at the software level. Its details outline abstract steps to ensure the safety of each component of the software. These procedures require the definition of functional requirements, resource usage, and the potential causes of software failure for any reason. The standards require that any "safe" component be tested under normal operational conditions with relevant faults inserted into the system. During the course of these tests, it is expected that faults, if present, are handled gracefully.

These sections of ISO 26262 are relevant to the project since they describe the design process that all companies seeking to develop passenger vehicles needs to follow. The recommendations in these sections will be applied to the BPMP firmware, which our team is working on. Understanding these processes gives us a better understanding of what the entire team's process will entail.

2.3.2 Automotive SPICE Process Reference Model

The Automotive Software Process Improvement and Capability Determination (SPICE) process reference model is a set of technical standards documents for software development processes related to automotive vehicles (VDA QMC Working Group 13 / Automotive SIG, n.d.). The standards are non-specific so that organizations following the standard are free to use any tools and processes, provided the basic requirements of the standards are met.

The relevant sections of the Automotive SPICE process reference model cover system-level development, software development, quality assurance, and project management (VDA QMC Working Group 13 / Automotive SIG, n.d.). The ultimate goal

of the processes described in the document is to ensure thorough documentation and complete understanding by its authors of the system being developed. This documentation should be thorough enough such that a third party can verify that the organization is following the Automotive SPICE guidelines and procedures. Additionally, it must be possible to track the realization of a system to their written requirements, and from written requirements back to the actual system. Any updates to that system should continue to match the requirements for older versions of the same system.

Development of provably safe software is done using the engineering verification and validation model shown in Figure 2.1. In this model, every component of the system is designed based on system-level and functional requirements before it is developed, and tested as a unit, as a component of the entire system, and from end to end before it is integrated. This strict process is designed to reduce the probability of faults that could cause catastrophic failures.

Much like ISO 26262, understanding the guidelines present in Automotive SPICE is important so that our team understands how the BPMP team needs to function in order to achieve its objective of ensuring software safety. In particular, these guidelines motivate the use of code review tools such as Gerrit to keep track of any and all changes that are being made to the BPMP codebase.

2.3.3 Software Safety Processes within Agile Software Development

Software teams at NVIDIA follow an agile software development model. This model suggests that requirements and their solutions should be developed in tandem, which results in an iteratively developed software product that adds features over time. This is often done out of necessity as it can be impossible to determine the full set of requirements for a piece of software at the onset of development. However, the

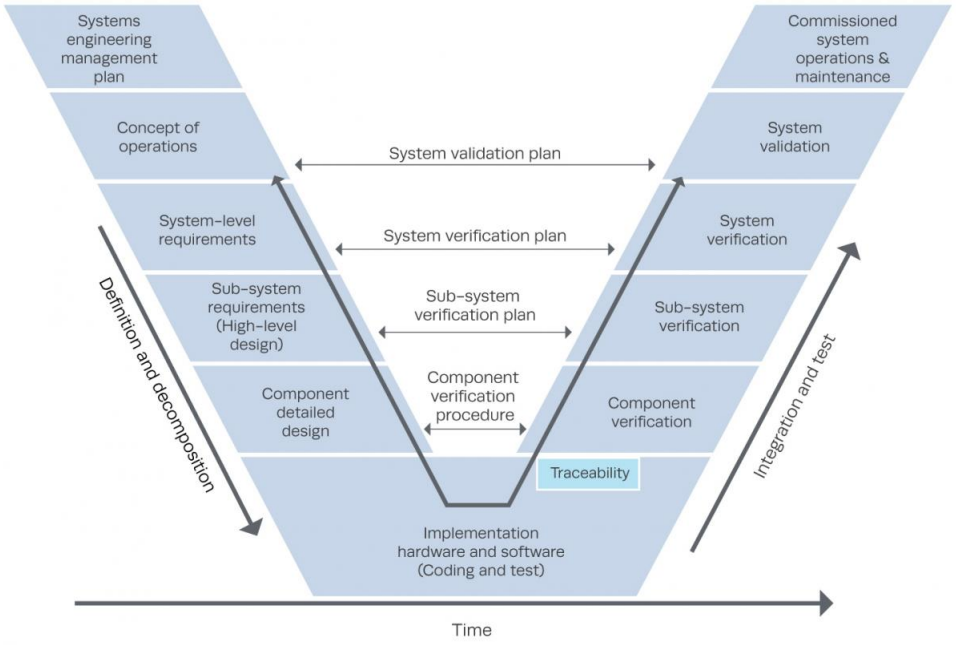

Figure 2.1: The engineering V-model.

agile development model stands in contrast to the V-model, which requires that requirements be clearly defined before any development can be done. This means that agile software development alone is insufficient to ensure functional safety to the levels that the BPMP firmware project requires.

The V-model shown in Figure 2.1 shows time increasing from left to right (Donahue & Van Schalkwyk, 2018). To achieve the same processes in an agile development model, time increases from top to bottom. The first step after the software has been created is to write software design documents for each functional unit, and to verify them using unit tests. The next step is to write software architecture documents and functional tests for each system above it, and so on. This continues until the entire BPMP firmware is covered by a sweeping set of architectural and unit design documents, and tests that range in scope from unit tests to end-to-end tests. If any new requirements are developed, or if any verification fails, the agile process will

begin anew to accommodate those changes.

The V-model is held up as a standard of how the development of safe software *should* be done. In NVIDIA's case, the BPMP firmware already exists, so it is necessary to create design documents and unit tests after the fact to ensure compliance with the Automotive SPICE process.

2.4 Code Standards

The BPMP firmware must adhere to project-specific code standards such as MISRA C:2012 and functional testing completeness. The goal of these code standards is to create uniform development habits among BPMP software engineers so that reading, examining, and maintaining code becomes easier. To accomplish this, NVIDIA has integrated a static analysis tool and unit testing framework to ensure code compliance with these standards.

2.4.1 Functional Testing Completeness

The BPMP team is interested in three types of coverage when testing for completeness of functional testing: line coverage, branch coverage, requirements coverage. Line and branch coverage are concerned with the source code itself; line coverage checks whether each line of code in a codebase is covered by the execution of a set of functional tests, and branch coverage tests whether each branch of execution is covered by the same. Requirements coverage operates on a higher level, and is concerned whether each functional requirement (as described in a software design document) is covered by the set of functional tests.

When developing these tests, software safety guidelines suggest that each component of the system be fully verified. As a result, the BPMP team has imposed a re-

quirement upon the set of functional tests that they exhibit complete line and branch coverage. They also require that each individual test can be cross-referenced with a specific functional requirement in a software design document. This allows each functional requirement to be easily verified, increasing the team's confidence in the safety of the BPMP firmware.

Ensuring the functional completeness of unit tests is necessary to ensure that the software works exactly the way it is expected to. Part of our team's project was to write unit tests to increase the line and branch coverage of the BPMP firmware's entire unit test suite.

2.4.2 MISRA C:2012

MISRA C:2012 is a code standard for the C programming language that seeks to eliminate the presence of undefined, unspecified, or implementation-defined behavior and to increase the readability of code by removing ambiguities. It was originally developed by the UK-based Motor Industry Reliability Association (MISRA, 2013) for use in software that runs in automobiles, but has also been adopted by many embedded software projects that require verifiable software safety. By adopting the standard, NVIDIA hopes to minimize the probability of its code exhibiting a fault due to undefined behavior, or of programmer error due to obtuse code. Either of these issues could potentially result in a software failure.

MISRA C:2012's guidelines consist of directives, which are high-level guidelines that apply to the codebase as a whole, and rules, which apply to individual lines or sections of code (MISRA, 2013). All rule violations should be detectable by a competent static analysis tool (such as Coverity, which NVIDIA uses). Guidelines are also classified as either mandatory, required (where a formal deviation report is needed to justify breaking it), and advisory (which should still be documented if broken). Di-

rectives are never classified as mandatory due to their large scope, but any guideline can be re-categorized by an organization to be more restrictive than the standard specifies.

To ensure compliance to the standard, NVIDIA has appointed an internal review board for MISRA compliance that is responsible for assessing every deviation that is filed and determining its validity. Any deviation report must prove that the code that causes the violation cannot cause a catastrophic failure if present, and that any undefined behavior as a result of it is minimized. Ultimately, the review board has full control over which deviation reports are accepted. The BPMP firmware team also runs a weekly scan of the entire codebase to count all MISRA violations (except for the ones with accepted deviation reports), and has developed a tool to scan for MISRA violations in individual files on demand in order to simplify their removal.

A particularly disruptive guideline in the MISRA standard is the required directive 4.12, "Dynamic memory allocation shall not be used" (MISRA, 2013). This applies to the C standard library functions `malloc` and `free`, as well as any other third-party allocators. This directive is present because of all of the possibilities for undefined behavior that can come as a result: the ISO C standard does not specify what happens when memory that was not dynamically allocated is freed, when freed memory is accessed, when there is no more memory that can be allocated, and what the contents of freshly allocated memory are. To comply with this rule, NVIDIA engineers are required to develop other solutions to acquire a variable amount of memory, or to justify the use of dynamic memory allocation by proving that it cannot cause any ill effects in the code.

ISO 26262 requires that a specific code standard be followed by an organization during the development of the firmware. NVIDIA chose to follow the MISRA C:2012 guidelines partly because the standard was originally developed for automotive soft-

ware. Part of the team's project involved refactoring existing code to conform to MISRA's guidelines.

Chapter 3

Methodology

This chapter discusses the specific tasks assigned to the team as part of our project. Design-oriented tasks are discussed first; these include a reorganization of source code to better match the defined architecture of the BPMP firmware. Next, the report discusses the development of software design documents for each component of the firmware, which are necessary for following ISO 26262 guidelines. The next section describes implementation-oriented tasks; of those, we were only responsible for MISRA-cleaning individual source files and conducting peer reviews on code written by other members of our team. These changes are meant to reduce the likelihood of undefined behavior and programmer error, respectively. The final section covers the development of unit tests to verify that defined functional requirements of the BPMP firmware are satisfied.

3.1 Software Definition and Design

The V-model described in ISO 26262 and the Automotive SPICE process reference model, shown in Figure 2.1, requires the development of an architecture design that

describes the requirements of each sub-system of the entire software product (Donahue & Van Schalkwyk, 2018). While the ideal architecture of the BPMP firmware can be described informally by software architects and team leads, it is still necessary to document the actual architecture of the firmware. To accomplish this, the team decided to develop documentation on the ideal architecture of the BPMP firmware while simultaneously refactoring the code to match that architecture.

The true architecture of a large software project like the BPMP firmware is difficult to realize concretely. To aid in this task, NVIDIA acquired licenses for a static code analysis tool called SciTools Understand (SciTools, n.d.). One of its most powerful features is that it can generate a butterfly dependency graph for each source module. A butterfly dependency graph shows all modules that depend on a given source module, as well as all modules that are depended upon by that given source module. In the example shown in Figure 3.1, the module `services/debugfs-service` makes 703 references to the `lib` module, 0 references to the `startup` module, and 2 references to the `drivers` module. Likewise, the `drivers` module makes 417 references to `debugfs` interfaces.

SciTools Understand's knowledge of the source modules in a project depends on an externally generated architecture file, derived from Python scripts in the BPMP codebase. Two distinct architectures have been developed to describe the firmware. One is based on a hierarchical model of each source module: a source module that interfaces with an analog-to-digital converter would be named `drivers/adc` and would be present under a higher-level source module called `drivers`, which is shown in the dependency graph in Figure 3.1. The other model splits the BPMP codebase into four slices: three lettered ones that need to be safety-verified, and one called **Excluded** that does not, as it only contains unit tests, stubs, and debugging code that will never be deployed in production. It is in the interest of the entire

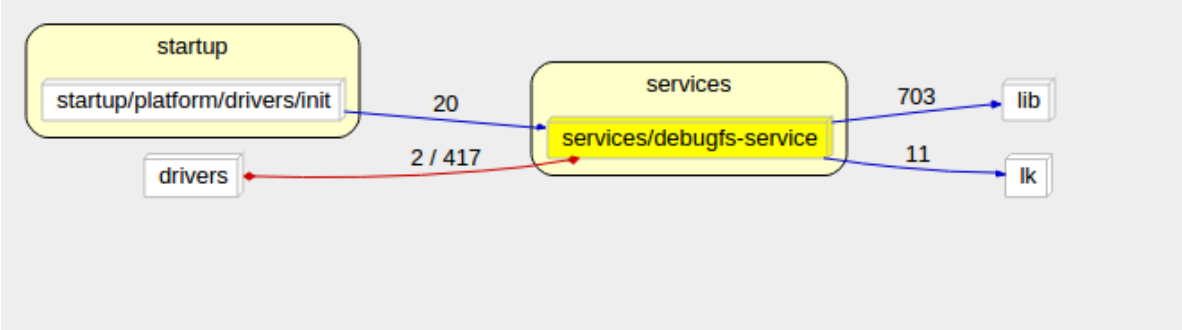

Figure 3.1: A butterfly dependency graph generated by SciTools Understand.

BPMP firmware team to place as much code in the **Excluded** slice as possible, since the code in that slice will not need to be MISRA-cleaned or properly verified.

3.1.1 Source Code Segregation

One major architecture change is the extraction of `debugfs` interfaces from the source code for every BPMP driver, and its consolidation in one directory outside of the drivers' source code. The `debugfs` service, as the name describes, is a debugging filesystem used for diagnosing issues with the hardware or software during testing. Each driver in the BPMP firmware contains a set of debugging routines to initialize `debugfs` and write data to the file system as needed. Since `debugfs` is not present in the firmware in production, it does not need to be subjected to the same test coverage and MISRA compliance standards that other production code does. However, each BPMP driver (which contains a `debugfs` interface) will be present in production and needs to be verified per the safety standards. Instead of needlessly MISRA-cleaning and unit testing an internal debugging framework, it was determined that `debugfs` needed to be extracted from each BPMP driver, placed into a different top-level module, and moved to the **Excluded** slice.

The first step of this process was to put source and header files containing ref-

erences to debugfs into a submodule of the same name. To do this, the team first determined which source files within each module reference the debugfs interface. Prior implementations of debugfs were present in their own source files, which made this process as simple as searching for files that contain the string debugfs or include the necessary header file lib/debugfs.h. Each source or header file that fit this description was placed in a sub-directory called debugfs, and a rules.mk file was created to turn that sub-directory into a proper module. Finally, the driver's top-level module was temporarily made to link to its debugfs submodule when debugfs was globally enabled.

The second step of this process was to move each driver's debugfs interface into a completely separate part of the source tree: for example, a debugfs interface located at drivers/adc/debugfs would be moved to srv/debugfs/adc. To accomplish this, a new Makefile was written for srv/debugfs that would only link a debugfs interface into the target binary if debugfs was globally enabled *and* the relevant driver was enabled. If debugfs was globally disabled, nothing would be linked; if the driver was disabled, then a stub interface would be linked instead, containing only a debugfs initialization subroutine with an empty body. To finish this step, every module inside srv/debugfs was placed inside a top-level module called debugfs-service, and relegated to the **Excluded** slice.

Following the completion of this step, the team used SciTools Understand to determine whether the migration of the debugfs interface was complete. Understand generated figure 3.1 when prompted about the dependencies of the debugfs service. The figure indicates that there were two references to a debugfs interface that were actively used by the code. Performing a recursive search using grep revealed many more that were declared but never used substantively[1]. The team then submitted

[1]Most of these turned out to be definitions of stub functions.

Figure 3.2: A butterfly dependency graph generated by SciTools Understand after the `debugfs` interface code was fully segregated.

a series of ten small-scale patches that collectively removed these references and placed them inside the `debugfs-service` module. Many of them simply involved the removal of a stub function from a header file and placing it inside a stub submodule of a driver's `debugfs` interface. The migration of other drivers, especially those related to runtime code profiling, was more involved, as it proved necessary at times to split the source files of those drivers to remove the `debugfs` interface inside them. At the conclusion of this final stage, SciTools Understand emitted the butterfly dependency graph shown in Figure 3.2, which revealed that no BPMP drivers reference `debugfs`, and a recursive search of source files revealed that the only textual reference to `debugfs` outside the startup code is present inside a comment.

3.1.2 Software Design Documentation

For compliance with ISO 26262, the BPMP team must create design documents for each software component (ISO, 2018). When documenting software design, the specifics of a module's functionality are separated into architecture design documents and module design documents, which are detailed below.

The BPMP team writes their documentation in a format called AsciiDoc, which al-

lows authors to write documents in plain text with basic formatting. A related tool, called AsciiDoctor, converts these files into HTML, PDF, and LaTeX formats, among others. In the BPMP firmware, documentation is placed in its own Git repository, which allows the team to keep track of documented components and how the documentation has changed. Each full build compiles the documentation into HTML and PDF formats and deploy it on NVIDIA's internal documentation website.

When developing architecture design documents for a module, it is necessary to list the modules that depend on it, as well as the modules that it depends on.[2] Architecture design documents also describe the functionality of the module as a whole, grouped by the source file containing that functionality. Additional content can be added as needed; if the module manipulates a data structure, the design document will contain a diagram that details how the data in that structure is used by each part of the firmware.

Module design documents provide an overview of the internal functional interface of the module. They list the details of each function exposed by the module, including the functions' possible return values, error codes, and parameters, an overview of the function's expected behavior and requirements, and a butterfly-dependency graph that shows its dependencies. This documentation allows developers to design unit tests that ensure that the functions do exactly what the requirements describe. The development of unit tests that follow this pattern reduces the probability that code containing a potentially disastrous fault will be deployed.

[2] SciTools Understand's butterfly-dependency graphs show this visually.

3.2 Software Implementation

Our team was also tasked with writing software to be added to the BPMP firmware codebase. We wrote unit tests for each specific module, and cleaned up MISRA violations in various source files throughout the BPMP software. All code submitted was subject to code reviews conducted by the other team members.

3.2.1 Code Review

When making changes to BPMP firmware, the assigned member created a new Git branch locally, and made one or more commits along that branch for certain milestones. When a change was ready to be reviewed, an interactive rebase would be performed to move the branch on top of `master` and squash the entire branch into one commit that could then be directly applied ("cherry-picked" in Git parlance) to the head of `master`. This commit would then be pushed to Gerrit. If further changes were needed because negative feedback was received during code review or the build failed, we would make any necessary corrections, amend the commit, and push again. Gerrit then notified other users that a new patch has been uploaded for the same commit.

Each commit submitted to Gerrit is automatically checked for common issues by a pair of automated processes running on a continuous integration server. The "Automatic Commit Validation User" process checks the contents of each patch to ensure that no binaries, executable files, or large files are pushed to the Git server. The `svc-bpmp` process cherry-picks the commit onto a copy of the `master` branch and attempt to build it. If the build succeeds, `svc-bpmp` adds a Verified flag to the commit.

Before involving NVIDIA staff, we sought code reviews from one another. When

reviewing code, we ensure that it matches NVIDIA's accepted style guidelines. During the code review, we also ensure that every modification has a reason behind it, so that anyone who reads the code can easily determine what it does. This usually results in "magic numbers" being replaced with constants, and the simplification of particularly obtuse code. A reviewer will add a **Verified** flag to the commit if it builds and tests successfully on his machine, and a code review flag between **-2** and **+2**. Our team was not allowed to give **+2** to code reviews; only a select few developers are capable of issuing a **+2** code review flag. When a commit is given a **+2** and at least one other person has reviewed the code, the commit is automatically cherry-picked and merged onto the `master` branch.

3.2.2 MISRA Violation Removal

Since the BPMP codebase was originally written without the MISRA C standard in mind, the BPMP firmware group needs to go through all of the source files and remove any violations that exist ("cleaning up" the codebase). Our team was tasked with removing MISRA violations in specific files in the BPMP codebase.

In order to clean MISRA violations in a given file, the first step is to run a script (called `misra-scan-file`) that enumerates every MISRA violation in that source file or any headers that it includes in an HTML report. For each violation, the report lists the ID of the rule that has been violated, a link to the line in the source file containing the violation, and the name of the function that contains the violation, if applicable. This scan has the ability to ignore a specific set of errors, which allows the BPMP firmware team to focus on cleaning a few errors at a time rather than try to clean all of them at once. Once changes are made, it is possible to generate a new report using the same script; each execution takes about two minutes for a 100-line source file. This process continues iteratively until the file is free of MISRA violations

or cleaned to a specific standard, after which the developer would submit the source file (and other associated changes) to Gerrit to be reviewed.

3.3 Software Verification by Unit Test Development

The Automotive SPICE process reference model suggests that test cases be written before the relevant functionality is implemented. However, the BPMP firmware was originally developed before NVIDIA knew it would be used in autonomous vehicles, and had sparse unit test coverage at the time the project began. To conform with the necessary safety processes, unit tests need to be written in such a way to cover all lines of code as well as all branches in the code as well.

The BPMP firmware's unit tests are developed using an internally developed unit testing library called `libut`. This library exposes functionality to assert whether two values are equal, and to mock interfaces for other components in the software, that the unit test will assume works as expected. Assertions are able to print a custom message to the console when a test fails, which makes it easy for developers to determine which assertion failed in a group of tests.

Device tree source files are used to store information about hardware components that are relevant to the operating system. They are used in both the Linux kernel and the BPMP firmware (Rowand, 2018). Modules that interact with device trees will be unit tested by writing device tree source files that cover every case of possible values. This made it necessary for the team to write device trees that, when loaded, will cause every line and branch of code to be executed. To integrate them with the firmware (for simulation and testing and also in production), each device tree source file is compiled into a binary blob that defines the device tree, and is embedded as read-only data inside the executable. The firmware interacts with the

binary blob using the interfaces provided by the third-party `libfdt` library.

Chapter 4

Results

Since our project was not fully self-contained, results are measured in terms of patches submitted or lines changed. This chapter begins with a table showing cumulative totals of work done through the project. We then display graphs that show our progress over time.

The data in the following sections were collected and placed into an SQLite database using a Python script. Pure SQL commands and other Python tooling was used to generate the cumulative and time series data presented in the following sections. The source code of all results tooling can be found in appendix A.

4.1 Work Totals

Table 4.1 details the cumulative work that our team has put into the BPMP safety project, showing the number of commits we submitted, how many of those were successfully merged into the actual codebase, and how many lines of code we changed in order to fulfill our goals.

Total Commits	60
Merged Commits	49
Unit Test Commits	16
Unit Tests Written	43
MISRA Cleanup Commits	18
Documentation Commits	3
Total Lines Added	5,736
Total Lines Deleted	1,814
Unit Test Lines Added	1,796
MISRA-compliant Lines Added	662
MISRA-violating Lines Removed	545
Work Days	36

Figure 4.1: Total work completed.

4.2 Work Accomplished over Time

Figure 4.2 displays the number of commits that our team has submitted since we started working on the BPMP safety project. The x-axis is the time in days since starting in January. The blue line comprises all of the code that has been submitted to Gerrit, while the red line shows how many of our commits have been merged into the codebase.

Figure 4.3 showcases the cumulative number of lines that were modified in each commit over the course of our project work. The x-axis also shows the time in days since work started on the project. The blue line specifically displays the number of lines that were added to the master branch while the red line shows the number of lines removed.

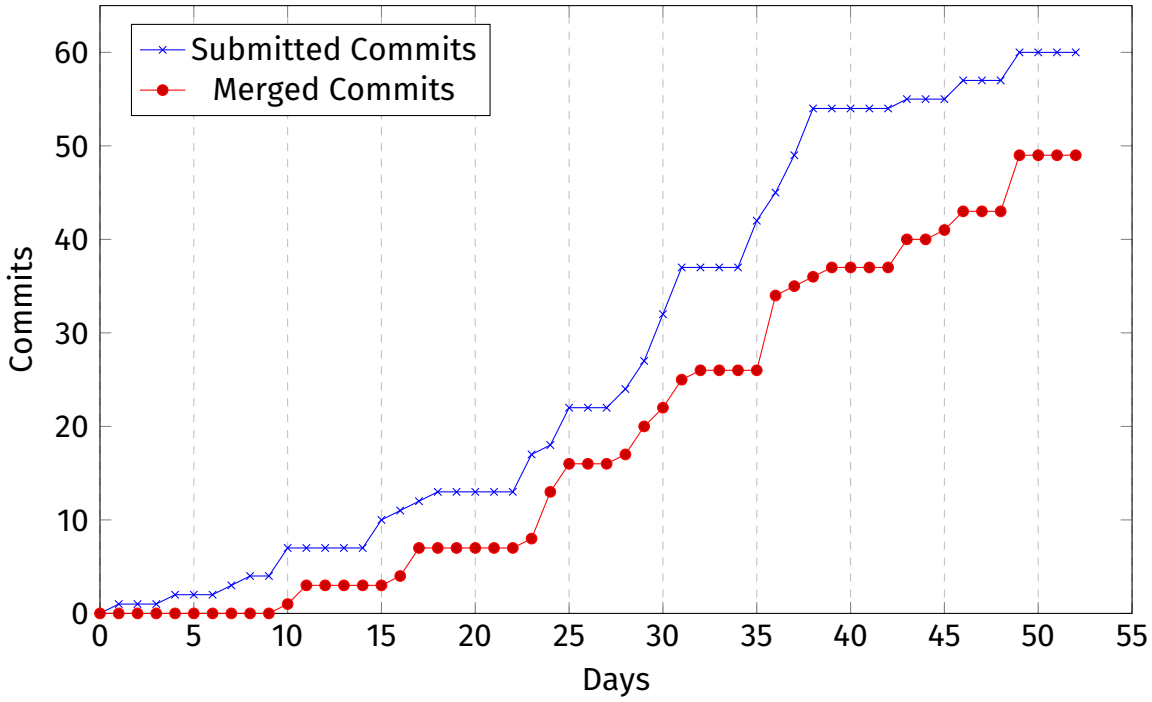

Figure 4.2: Submitted and merged commits over time.

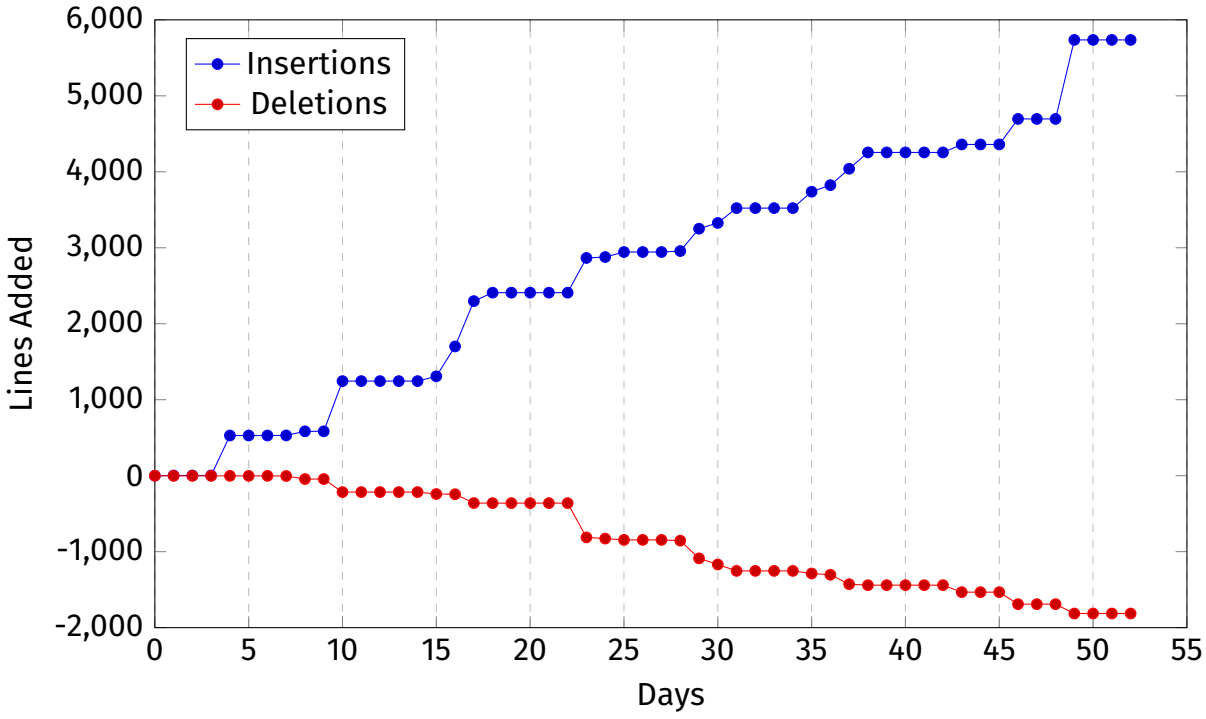

Figure 4.3: Line changes over time.

In summary, the team was able to resolve hundreds of MISRA violations in the BPMP codebase, write many unit tests to verify the functionality of the BPMP firmware, and develop detailed documentation on BPMP source modules. Additionally, the team was able to fully extract debugging interface code from BPMP drivers so that it would not have to be held up to the same safety standards as production code.

Chapter 5

Conclusion

Ensuring that software systems are safe is hard, requiring code to be well documented, fully tested, and reliable. These tasks are important as they have proven to reduce the probability of faults in software causing catastrophic failures in hardware systems. Software safety can be achieved through work prescribed by safety guidelines like ISO 26262 and Automotive SPICE, which require organizations to follow a specific development process when developing software for passenger vehicles. Ensuring that the systems used to power autonomous vehicles are safe will reduce the risk present to those who use them or exist around them.

Our team has achieved the primary goal of assisting the BPMP firmware group with achieving compliance with ISO 26262 and the Automotive SPICE safety guidelines, and code compliance with MISRA C:2012 standards. The team wrote unit tests, refactored BPMP drivers, cleaned MISRA violations across the codebase, and drafted software design documents for BPMP libraries. As displayed in the results section, our team made significant contributions to the BPMP firmware in all of these categories. By completing these tasks, our team has reduced the amount of work that the BPMP firmware group needs to complete for the safety project.

Chapter 6

Future Work

The following paragraphs briefly describe work that could be done to further BPMP software safety; specifically drafting software design documents, writing unit tests, and cleaning up MISRA violations from the BPMP codebase to make the firmware compliant.

The design documents written by our team are under review by more senior members of the BPMP team. These documents are important since NVIDIA plans on using these libraries in other projects in the Tegra software stack. Publishing these design documents may be beneficial for developers to reference when writing software that depends on those libraries. There are other software components that need to be documented in this fashion. Design documents must describe the libraries' purpose, its functional requirements (how the software responds to inputs), its dependencies, the interface it exposes to other software units, and how errors are handled internally. Separate documents must describe each file or unit within the module and its role therein.

There are other unit tests that need to be written to verify that the entire BPMP firmware functions as expected. Of the libraries the team wrote unit tests for, none of

them exhibit complete line and branch coverage yet; additionally, when more functional and safety requirements of these libraries are created, more unit tests ought to be developed that correspond to those written requirements. Achieving a direct relationship between written requirements and functional tests may make it easier to verify that each software unit performs the way that its design document claims it does.

Finally, MISRA violations need to be resolved in files throughout the BPMP codebase. Thousands of MISRA violations still exist in the BPMP codebase that either need to be removed, or waived by NVIDIA's internal safety committee. While some violations are relatively easy to fix, attention could be directed towards violations of more abstract and involved rules, like those the prevent the use of recursion or dynamic memory allocation. Removing these violations may decrease the likelihood that the firmware (written in the C programming language) would exhibit undefined behavior. Additionally, following these rules may increase the readability of the codebase and help others who are not familiar with it determine what the code is doing.

References

Daniel, C. (2018, Sep). *Signed, SHIELD, delivered: Our streaming media player gets its 20th upgrade.* NVIDIA Corporation. Retrieved February 5, 2019, from https://blogs.nvidia.com/blog/2018/09/10/shield-update-20/

Donahue, J., & Van Schalkwyk, I. (2018, July). WSDOT design manual. In (chap. 1050.03). Washington State Department of Transportation.

Franklin, D. (2017, Mar). *NVIDIA Jetson TX2 delivers twice the intelligence to the edge.* NVIDIA Corporation. Retrieved January 18, 2019, from https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/

Gerrit Code Review. (n.d.). *About Gerrit.* Gerrit Code Review. Retrieved January 31, 2019, from https://www.gerritcodereview.com/about.html

ISO. (2018). *Road vehicles – functional safety* (ISO No. 26262–1:2018). Geneva, Switzerland: International Organization for Standardization.

Karandikar, A. (2017, Sep). *NVIDIA gaming technology powers Nintendo Switch | NVIDIA blog.* NVIDIA Corporation. Retrieved January 24, 2019, from https://blogs.nvidia.com/blog/2016/10/20/nintendo-switch/

MISRA. (2013). *MISRA-C: 2012: guidelines for the use of the C language in critical systems.* Motor Industry Reliability Association.

NVIDIA. (2015, Dec). *Tegra boot flow | Tegra public app notes.* Santa Clara, CA: NVIDIA Corporation. Retrieved January 28, 2019, from https://http.download

.nvidia.com/tegra-public-appnotes/tegra-boot-flow.html

NVIDIA. (2017). Nvidia Tegra boot and power management processor (BPMP) [Computer software manual]. The Linux Kernel Archives. Retrieved February 13, 2019, from https://www.kernel.org/doc/Documentation/devicetree/bindings/firmware/nvidia%2Ctegra186-bpmp.txt

Rowand, F. (2018, Dec). *Device tree reference.* eLinux. Retrieved from https://elinux.org/Device_Tree_Reference

SciTools. (n.d.). *Understand: Visualize your code | scitools.com.* Scientific Toolworks, Inc. Retrieved February 6, 2019, from https://scitools.com/features/

Shapiro, D. (2016, Oct). *Tesla Motors' self-driving car "supercomputer" powered by NVIDIA DRIVE PX 2 technology.* NVIDIA Corporation. Retrieved February 14, 2019, from https://blogs.nvidia.com/blog/2016/10/20/tesla-motors-self-driving/

Smith, R., & Ho, J. (2015, Jan). *NVIDIA Tegra X1 preview & architecture analysis.* AnandTech. Retrieved January 24, 2019, from https://www.anandtech.com/show/8811/nvidia-tegra-x1-preview/4

Ubuntu. (n.d.). *Ubuntu release cycle.* Canonical Ltd. Retrieved January 31, 2019, from https://www.ubuntu.com/about/release-cycle

VDA QMC Working Group 13 / Automotive SIG. (n.d.). *Automotive SPICE process assessment / reference model.* (Version 3.1)

Appendix A

Results Tooling

A.1 Gerrit Scraper

This script scrapes the Gerrit Code Review server for data about patches submitted by the authors of this report. Since the cookie variable contains data irrelevant to the operation of this script, its value has been omitted.

```
#! /usr/bin/env python3

import json
import sqlite3
import subprocess

# This Gerrit endpoint works (*for now*)
url = "https://git-master.nvidia.com/r/changes/?q=owner:tejasr@nvidia.com
   &q=owner:rharrison@nvidia.com&q=owner:gdeva@nvidia.com&O=881"
cookie = "redacted"

cmd_output, _ = subprocess.Popen(["curl",
```

```
  url,
  "-H", "User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:65.0)
    Gecko/20100101 Firefox/65.0",
  "-H", "Accept: */*",
  "-H", "Accept-Language: en-US,en;q=0.5",
  "-H", "--compressed",
  "-H", "Referer: https://git-master.nvidia.com/r/",
  "-H", "DNT: 1",
  "-H", "Connection: keep-alive",
  "-H", cookie], stdout = subprocess.PIPE, stderr = subprocess.PIPE).communicate()
raw_commits = json.loads(cmd_output.decode("utf-8")[4:])
raw_commits = [item for sublist in raw_commits for item in sublist]

patch_db = sqlite3.connect("commits.db")
patch_cur = patch_db.cursor()
patch_cur.execute("CREATE TABLE IF NOT EXISTS PATCHES (ID INTEGER PRIMARY KEY,
OWNER TEXT, CREATED TEXT, FINISHED TEXT,
SUBJECT TEXT, PROJECT TEXT, INSERTIONS INTEGER,
DELETIONS INTEGER, MERGED INTEGER)")
patch_db.commit()

for c in raw_commits:
    if c["status"] != "ABANDONED":
        patch_cur.execute("INSERT INTO PATCHES VALUES
                (?, ?, ?, ?, ?, ?, ?, ?, ?)",
          [c["_number"], c["owner"]["username"],
           c["created"][:23],
           None if c.get("updated") is None else c["updated"][:23],
```

```
            c["subject"], c["project"], c["insertions"], c["deletions"],
            1 if c["status"] == "MERGED" else 0])
patch_db.commit()
```

A.2 Database Scraper

This script scrapes the SQLite database containing commit data, and writes the results of the given query in a format that the chart renderer can interpret. This version of the script displays merged commits over time (see section A.3 for other statements that can be put in its place).

```
#! /usr/bin/env python3

import datetime
import sqlite3

curr_date = datetime.datetime(2019, 1, 8)
rollup_commits = []
db = sqlite3.connect("commits.db")
cur = db.cursor()
while curr_date <= datetime.datetime(2019, 3, 1):
    cur.execute("SELECT COUNT(*) FROM PATCHES WHERE MERGED = 1 AND
        FINISHED < ?", [curr_date + datetime.timedelta(hours = 8)])
    this_date_commits = cur.fetchone()
    rollup_commits.append(this_date_commits)
    curr_date += datetime.timedelta(days = 1)

print(" ".join(str((a, *b)) for a, b in zip(range(len(rollup_commits)),
```

```
    rollup_commits)))
```

A.3 SQL Statements for Chart Data Retrieval

The following statements can be run directly against the database:

- Total Commits: `SELECT COUNT(*) FROM PATCHES`
- Merged Commits: `SELECT COUNT(*) FROM PATCHES WHERE MERGED = 1`
- Unit Test Commits: `SELECT COUNT(*) FROM PATCHES WHERE SUBJECT LIKE "%unit%"`
- MISRA Cleanup Commits: `SELECT COUNT(*) FROM PATCHES WHERE SUBJECT LIKE "%MISRA%"`
- Documentation Commits: `SELECT COUNT(*) FROM PATCHES WHERE PROJECT = "tegra/bpmp/doc"`
- To determine the total number of inserted or removed lines, we select `SUM(INSERTIONS)` or `SUM(DELETIONS)` instead of `COUNT(*)`.

The following statements need to be run using the database scraper script found in section A.2.

- Submitted Commits over Time: `SELECT COUNT(*) FROM PATCHES WHERE CREATED < (date)`
- Merged Commits over Time: `SELECT COUNT(*) FROM PATCHES WHERE MERGED = 1 AND FINISHED < (date)`
- Insertions over Time: `SELECT COALESCE(SUM(INSERTIONS), 0) FROM PATCHES WHERE CREATED < (date)`

- Deletions over Time: `SELECT COALESCE(-SUM(DELETIONS), 0) FROM PATCHES WHERE CREATED < (date)`