Enhancing Network Security through Vulnerability Monitoring

Ryan Williams[1], Anthony Gavazzi[1], and Engin Kirda[1]

Northeastern University, Boston MA, USA
{williams.ry,gavazzi.a,e.kirda}@northeastern.edu

Abstract. In modern cyberattacks, adversaries no longer focus solely on individual computer systems but instead establish an initial foothold within a company's network, advancing through compromised assets in a process known as lateral movement. Detecting lateral movement is challenging due to diverse infection vectors, making network traffic monitoring prone to false positives and negatives. Security patches, while crucial, can create a false sense of security. To address these issues, we introduce PatchCanary, a framework for augmenting source patches for CVE-identified vulnerabilities, allowing precise monitoring of modified functions. We propose the idea of "patch and monitor" as a new approach to vulnerability patching, enhancing lateral movement attack detection. Evaluation on 108 CVEs across 75 real-world programs demonstrates PatchCanary's capability to automatically augment source patches for 95.9% of CVE-triggering paths while incurring a minimal 712ms compile-time overhead, on average.

Keywords: Intrusion Detection · Network Security · Program Repair.

1 Introduction

Today, in a typical attack campaign, bad actors often have a concrete security-sensitive objective in mind, such as accessing a top developer's machine to steal a popular project's source code, accessing all of an important executive's files, or reading a database that stores customer credit card data [16]. While such data breaches often start with a single compromised system in an organization, the initial compromised asset is usually not the attackers' ultimate destination. Rather, after breaking into a web server, email account, employee device, or any other low-value starting location, the attackers will move "laterally" from the initial cyber "bridgehead" (i.e., foothold) that they have established to reach their intended target, or to opportunistically locate a target that is of value.

Lateral movement is when attackers acquire access to an asset within a network and are then able to spread their reach from that asset to others within the same environment. Today, the initial compromise itself in an organization seldom causes significant damage [49]. As a result, if an organization's security team can detect the lateral movement before the attackers are able to reach a more security-sensitive target, the data breach can potentially be mitigated.

Unfortunately, advanced attacks succeed because current security controls lack the ability to detect the malicious activity as it moves laterally across a network.

A typical organization's network has a security perimeter (e.g., a firewall or a security monitor) that separates and defines what is "inside" and what is "outside" the organization with respect to its security policies. Assets that are outside the security perimeter are called the "top half," and assets that are inside are called the "lower half." Hence, in order to compromise an asset within the organization, an attacker must first move vertically. That is, the initial attack needs to occur from the outside, and an asset that is inside needs to be compromised. This direction of the attack is often called North-South traffic. Once the attacker has established a bridgehead, however, they can now move laterally (or horizontally) within the network to reach their objective. This direction of the attack is often called East-West traffic.

In the attacks that are observed today, there are two main techniques that a threat actor uses to move laterally [37]. The first approach consists of the attacker stealing credentials belonging to unsuspecting users, and then using them to move laterally within the organization. In the second approach, the attacker deploys internal scanning to discover the network topology around the initial established bridgehead. Typically, the attacker scans for open ports that are listening for incoming traffic, and attempts to identify network services that suffer from (often known) vulnerabilities. Once a vulnerable network service has been discovered (e.g., a vulnerable internal print spool service), the attacker can exploit this weakness to move laterally within the organization and compromise another asset.

Because of the substantial consequences posed by data breaches, to date, there has been much research in the area of intrusion detection and prevention. These techniques focus on learning the characteristics of attacks and identifying similar attacks in the future [11, 33, 39, 45, 52], or learning what legitimate activity looks like and flagging anything that does not match what was learned as malicious [12,13,34]. In contrast, in this paper, we introduce a novel, deception-based approach that aims to detect vulnerability-based lateral movement attempts whenever an attacker attempts to compromise a vulnerability on a host that has already been patched.

When new CVEs (Common Vulnerabilities and Exposures) are disclosed, network managers and system administrators typically apply an available patch and then stop worrying about the initial flaw so that they can focus on other tasks at hand [30]. However, even if a vulnerability is patched and cannot be compromised anymore during a lateral movement attempt, the attacker may still remain undetected on the network, can wait, and can try other vectors of compromise including other vulnerabilities that exist in the organization. Thus, although patching a vulnerability is important for preventing a potential attack, it does not contribute at all to the detection of any attempts to compromise the vulnerability.

To bridge this gap and allow vulnerability patching to become a powerful contributor to the detection of lateral movement attacks, we propose the novel

idea of "patch and monitor" as an alternative to the traditional mindset of "patch and move on." That is, besides patching the vulnerability, we propose an automated approach that also inserts code to monitor the section of vulnerable code that was disclosed by a given CVE, and that allows the network manager or system administrator to *detect* when an attacker is attempting to compromise this patched vulnerability as an early warning system for lateral movement within the organization.

This work makes the following contributions:

- We develop PatchCanary, a novel system to semi-automatically generate and insert monitors to augment source patches;
- We propose the novel idea of "patch and monitor" as an alternative to the traditional "patch and move on" paradigm;
- We systematically evaluate PatchCanary on 75 real-world programs and 108 known CVEs. In our evaluation, we find that PatchCanary is able to successfully report potential indicators of compromise on patched vulnerabilities while incurring minimal overhead.

2 Background

While most existing security tools focus on detecting attacks coming from outside of a *security perimeter*, or monitor a specific type of artifact such as network traffic [32,45,52], in contrast, we aim to provide a finer granularity of control over monitoring systems and the vulnerability-based attack that is being launched. By providing a novel approach of monitoring individual functions in a program, we can tell how an attack is being attempted while being agnostic in terms of the source of compromise. For example, although monitoring an Apache web server for any anomalous traffic may provide insights on external actors attempting to gain access to the system, monitoring specific functions in Apache that were found to be vulnerable previously, and that were patched, provides a powerful method to detect indicators of a larger compromise [16]. In our approach, because we monitor individual functions, the overhead incurred is low enough (see §5) to consider all inputs as untrusted, regardless of source.

In practice, when patches are released, the most information provided along with it is a changelog that states what the patch fixes. Once the patch is applied, the effect–if any–is opaque to the user [54]. By inserting monitors, however, we are also provided with some insight as to the efficacy of the patch that was provided. If the patch came from an untrusted source, or if the network manager is simply interested in knowing if the patch is actually blocking any attempted exploitation, the monitors will provide that level of information.

While other detection systems may rely on classifying anomalous behaviors [7,47] that are all prone to false positives, PatchCanary is able to monitor functions and their parameters concretely. Hence, we can insert monitors into a system to look specifically for executions that pass input that is known to trigger a vulnerability from a CVE. This also means that the incurred overhead

of running systems with monitors that were inserted will be negligible compared to running machine learning-based solutions, or an intrusion detection system (as we show in §5). Because PatchCanary is implemented as a compiler pass in Clang, the overhead is confined to this monitor injection step.

2.1 Threat Model

We consider two threat models that both include an attacker who intends to compromise a high-value asset on a network. Note that this could be an attack from outside the security perimeter, or from within the network. PatchCanary operates agnostic to the source of the attack, as we intend to insert monitors that are not only looking for potential breaches, but also internal lateral movement.

In the first threat model, an attacker armed with a standardized exploit, scans the Internet for vulnerable devices. The attacker may use a vulnerability search engine such as Shodan [3], or randomly scan the IP address space. The attacker commandeers any device that replies to their probes and is vulnerable to the exploit, which can then be used as a foothold. In the second threat model, we consider an attacker who already has access to a node on the local network. In this instance, the attack would be more difficult to detect with traditional methods because there are no typical indicators of compromise. Instead, the signal the attacker generates may resemble normal traffic if it is passing between two trusted nodes on a network, as opposed to coming from a source outside the security perimeter.

In both threat models, the attacker most likely does not need to directly compromise every device; compromised devices may in turn become vectors of infection, as it is common in Internet worms and IoT botnets. The source of the attack is less important in our use case as we are more interested in first detecting a signal that may indicate not only compromise, but lateral movement. With both of our threat models, we consider any input to functions with monitors to be *untrusted*. This allows for detection of malicious input even from an otherwise trusted source.

2.2 Objectives and Goals

PatchCanary's goal is to provide a foundation to augment publicly-available patches with simple monitoring functionality. It explicitly aims to provide feedback on the efficacy of potentially untrusted patches, and provide early indicators of compromise regardless of the source. The purpose is to move from the current traditional mindset of "patch and move on" to our proposed "patch and monitor". We do not claim that PatchCanary captures data flows to every possible vulnerability, nor that it cannot be bypassed in individual cases by a skilled, motivated attacker. Its defined goal is to provide a mechanism for determining the efficacy of applied patches, and tracking inputs to monitor for early signs of compromise and lateral movement.

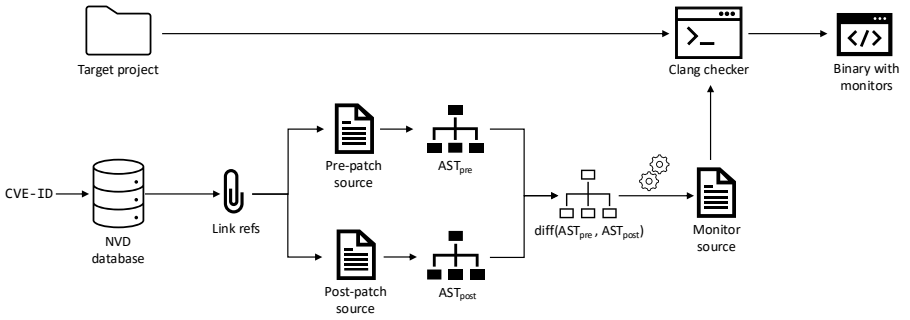

Fig. 1. High-level overview of PatchCanary's workflow where the two inputs are a target application and a CVE-ID corresponding to a vulnerability the analyst is interested in generating a monitor for.

3 System Workflow and Design

3.1 System Overview

In this section, we present an overview of PatchCanary. Figure 1 shows an overview of PatchCanary's workflow. At a high-level, PatchCanary is implemented as a Clang plugin to be used as a standard pass at compilation time, and as a LibTooling-based [4] standalone tool that can be used to perform analyses without full compilation. We chose to implement PatchCanary at this level due to the flexibility that the Clang API provides for performing source-level transformations using the underlying abstract syntax tree (AST).

Functionally, PatchCanary first looks up CVE disclosures for the target program, $\mathcal{P}$, and finds those that have a source patch file linked. Next, we create a set, $\mathcal{F}$, of functions that are modified in the patch file. This is necessary for knowing where to insert the monitors later. The next step is *trigger inference*, which aims to determine what input(s) will trigger the vulnerability disclosed in the CVE. This information is then used to generate a monitor that checks for the given critical input(s), and reports when there was an attempted exploit. The monitors are then inserted into the functions that were patched, and compilation continues to produce an augmented, patched program, $\mathcal{P}'$.

3.2 Patch Lookup

The first step in PatchCanary's workflow is to find a patch for a CVE that is of interest for monitoring. Given a target CVE, PatchCanary looks up details assigned to it using cve-search [1], which allows for efficient, local queries

Table 1. Parameters for Gathering Target Vulnerabilities

Parameter	Description
cvss	>= 7.0
exploitabilityScore	>= 8.0
impactScore	>= 6.0
vulnerable_product	¬(cpe:2.3:o ∧ cpe:2.3:h)
access.vector	NETWORK
access.complexity	LOW
access.authentication	NONE
references	Includes URLs

for known CVEs. PatchCanary then parses the CVE data in the output of cve-search for CVEs that have patches linked through version control systems (e.g., GitHub), pulls the patches via their commit hash, and saves them locally. For our evaluation, we filtered for CVEs that had a higher risk associated with exploitation (e.g., buffer overflows, remote code execution, etc.). This is found using the exploitability metrics provided by NIST vulnerability scoring system (CVSS). We also only look at patches for codebases that are C/C++-based as our tool is built on Clang. This resulted in targeting a total of **108** CVEs across **75** applications. The details of the query parameters used in cve-search are shown in Table 1. After obtaining the patch associated with the target program, PatchCanary parses the patch file to find which functions are modified in the patch, and saves them in a set, $\mathcal{F}$, which is used in the subsequent steps.

3.3 Patch Semantic Parsing

This step involves parsing the pre-patch and post-patch source files, generating their Abstract Syntax Trees (ASTs), and computing the differences between these ASTs to find the constraints introduced by the patch. The process begins by parsing both the pre-patch and post-patch versions of the source code to generate their respective ASTs. The ASTs provide a structural representation of the source code, highlighting the syntactic elements and their hierarchical relationships. By comparing the pre-patch and post-patch ASTs, we can compute the differences (diffs) that pertain to the exact changes introduced by the patch. Once the AST diffs are computed, we analyze these differences to extract the relevant constraints. These constraints typically correspond to new conditions, type modifications, or other logical changes added by the patch. By encoding these constraints symbolically using a tool such as Z3, we can formalize the logical conditions that the patch introduces. Next, the extracted constraints are used to generate the necessary monitoring conditions that will be inserted into the patched functions.

Algorithm 1 Patch Monitor Generation

Input: Source of pre-patch and post-patch source files
Output: Binary with inserted monitors

Input: AST_{pre}, AST_{post} ▷ Abstract Syntax Trees
`diffs` ← computeDiffs(AST_{pre}, AST_{post})
for each `diff` in `diffs` **do**
    if `diff.type` == ConditionAddition **then**
        condition ← extractCondition(`diff`)
        if *condition* == semanticInvariant(AST_{pre}, AST_{post}) **then**
            monitor ← generateDirectMonitor(*condition*)
        else
            inverse ← generateInverseCondition(*condition*)
            monitor ← generateMonitor(*inverse*)
    else if `diff.type` == TypeModification **then**
        typeChange ← extractTypeChange(`diff`)
        monitor ← generateTypeMonitor(*typeChange*)
    else if `diff.type` == Modification **then**
        modification ← extractModification(`diff`)
        monitor ← generateModificationMonitor(*modification*)
    else
        `diff` ← `"<unknown>"`
    insertMonitor(*monitor*, `diff.location`)
compileBinaryWithMonitors()

3.4 Monitor Generation

The monitor generation phase involves creating runtime monitors that can detect potential exploit attempts of the patched vulnerabilities. Our approach leverages symbolic execution to derive the necessary conditions from the patched code. The encoding of the vulnerability condition as constraints is then passed to the Z3 theorem prover to find a set of input values that satisfy these constraints.

Symbolic Encoding. For each identified change, we encode the corresponding patched code segment as symbolic constraints using a symbolic execution engine. These constraints represent the logical conditions introduced by the patch.

Condition Generation. Based on the symbolic constraints, we generate the conditions that need to be monitored at runtime. If a condition addition is detected, we generate both the direct condition and its inverse. For type modifications and other changes, appropriate monitoring conditions are derived.

When the theorem prover returns `satisfiable`, we can then return the constraints and concrete value(s) that were used. This ensures that the runtime monitors are precise and can effectively detect attempts to exploit the vulnerabilities addressed by the patches. The steps of monitor generation are outlined in Algorithm 1.

```
1 int test(int a, int b)
2 {
3   return a + b;
4 }
```

```
1 int test(int a, int b)
2 {
3   if (a < b)
4   {
5     return a + b;
6   }
7 }
```

Fig. 2. Example of a simple patch that will be detected via the semantic parsing and used as conditions in the monitor.

To illustrate this process, consider the example in Figure 2. In this example, the patch introduces a new condition: `if (a < b)`. The condition `(a < b)` is encoded as a symbolic constraint using Z3. The constraint represents the logical condition added by the patch. Both the direct condition and its inverse are generated from the symbolic constraint. $DirectCondition : (< a\ b)$, $InverseCondition : (\geq a\ b)$. The genrated inverse monitor is now able to detect an attempted exploit of the patched vulnerability, and is ready for insertion into the patched function.

3.5 Monitor Insertion

Finally, given a program $\mathcal{P}$, the set of target functions, $\mathcal{F}$, and the monitors, $\mathcal{M}$, PatchCanary is able to insert monitors inside the functions modified in the patch during compilation time, outputting an augmented, patched program, $\mathcal{P}'$. When inserting the augmented source patch for $\mathcal{P}$ that takes input y, we require that $\mathcal{P}'$ satisfies the following property: $\forall y, \mathcal{P}(y) = \mathcal{P}'(y)$. That is, the monitors we insert do not alter the control flow of the program. Instead, they passively monitor, and report alerts to the administrator.

Once we have generated the necessary monitors, $\mathcal{M}$, PatchCanary parses the AST of the program using our Clang checker. PatchCanary's compiler passes statically traverse the AST and find the target functions, $f \in \mathcal{P}$, along with their respective monitor(s), $\mathcal{M}_f$, that we wish to insert. Once it has the location of those function definitions, PatchCanary uses the program's AST to find an appropriate insertion point within the function, $\mathcal{P}_f$, for the monitor code generated in the previous step.

4 Implementation

Our PatchCanary prototype is built as a set of Clang compiler passes, along with a standalone, LibTooling-based tool for a simpler user interface. The monitor generation component utilizes Z3 [38] for finding the conditions that trigger a vulnerability. Our patch lookup component is based on *cve-search* [1], which allows us to perform more complex queries on CVEs from the NVD database locally. Building PatchCanary on Clang and LibTooling [4] provides us access

to a powerful set of APIs for analyzing and modifying source files in the C language family. Using this framework allows us to create PatchCanary without the need for re-implementing various functionalities like all our operations on a program's AST.

5 Evaluation

To evaluate the prototype of PatchCanary, we conducted: (*i*) an evaluation on the completeness of monitor generation (§5.2); (*ii*) tests to show the correctness and usability of the augmented patches generated including a real-world use case deployed on a production system (§5.3); (*iii*) performance measurements to show the practicality of using PatchCanary (§5.4), and (*iv*) case studies detailing monitor generation for specific CVEs (§5.5).

5.1 Experimental Setup

For our evaluation, we aimed to select a diverse and representative set of vulnerabilities. We focused on the C language family due to its widespread use and the high prevalence of vulnerabilities in C-based applications. Our dataset was automatically collected from the National Vulnerability Database (NVD) [5], where we filtered out vulnerabilities that require specific devices to trigger as well as those whose behaviors cannot be directly observed. We further concentrated our dataset based on Common Weakness Enumerations (CWEs). The focus was on CWEs with well-known triggers, ensuring a meaningful and consistent evaluation of PatchCanary's effectiveness. From this, we selected 108 real-world vulnerabilities to test the efficacy of PatchCanary. The vulnerabilities are from 75 applications which include media encoding libraries, messaging systems, PHP, and the Linux kernel.

All experiments below were performed on an Ubuntu 20.04 workstation with a quad-core Intel i7 @ 3.00GHz and 16GB of RAM.

5.2 Monitor Generation Completeness

To evaluate the effectiveness of our monitor generation process, we analyzed the completeness of the monitors generated for the 108 example CVEs. We focused on two key metrics:

- **Successful Monitors**: Monitors that were fully generated and able to detect potential exploit attempts;
- **Unknown Monitors**: Monitors that were partially generated or incomplete where a condition and/or variable to monitor could not be inferred.

We collected data on the number of successful and unknown monitors for each instrumented file. Across the 108 vulnerable projects targeted, PatchCanary generated a total of 782 monitors. This is because we consider a monitor to be

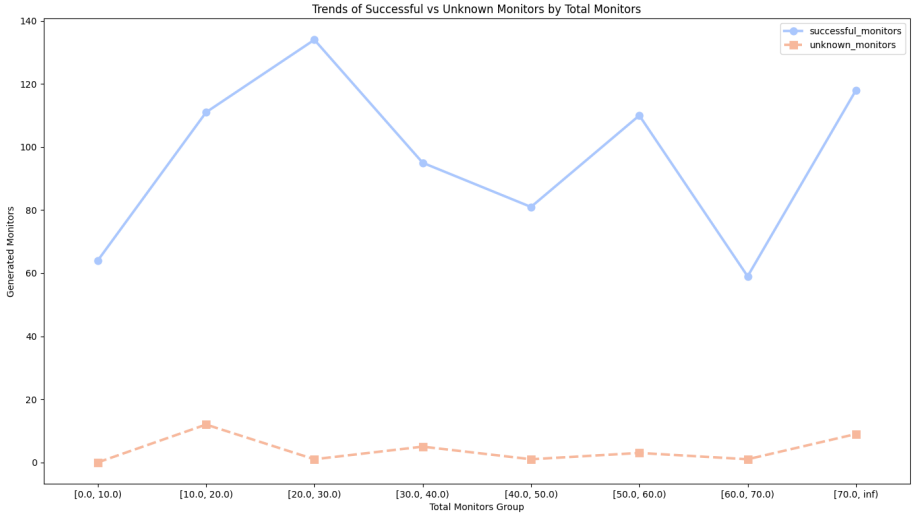

Fig. 3. Trends of successful versus unknown monitors grouped by the number of monitors needed for a given patch. E.g., for a CVE that required 100+ monitors for a patch, we have 9 cases where we are unsuccessful in generating the given monitor.

each inserted code block that detects a given pattern. In the case of a simple patch, we may see only one monitor; whereas a more complex one may have hundreds. Of the 782 total monitors, 750 of them were complete and the remaining 32 were unable to be inferred. That is, PatchCanary was able to generate source-level monitors for 95.9% of the target CVEs. The overall distribution of successful and unknown monitors as well as their relation to patch complexity is shown in Figure 3.

5.3 Augmented Patch Correctness

For all of the CVEs that we evaluated against, we require an evaluation on the efficacy of the augmented patches. The program that is now patched with monitoring functionality, $\mathcal{P}'$, must meet the following two properties: (i) any triggers that the original patch would defend against must remain unmodified; and (ii) when an input is passed that would otherwise trigger the vulnerability, the monitor must report the blocked attempt.

To test these properties, we compiled all of our codebases with the augmented patches, $\mathcal{P}'$. Next, we manually verified that the unpatched codebase, $\mathcal{P}$, was indeed exploitable with the triggering inputs. We then took the known inputs that trigger the given CVEs for each target, and manually attempted to trigger the vulnerability in $\mathcal{P}'$. We consider an augmented patch correct only when these tests are passed. In all of the test cases where a monitor was generated, the vulnerability described in the CVE was no longer exploitable, and on each

of the attempts, a report was logged from the monitors. Thus, our evaluation found that all generated patches are correct.

To illustrate real-world efficacy, we compiled a patched version of OpenSSH where we monitored the functions modified for the CVE-2021-28041 patch. We ran `ssh-agent` on a developer workstation for two weeks without any reported errors or issues from the users. Over the course of this two week-long experiment, we ran `ssh-agent` with various keys and agent forwarding, to attempt to exercise as many code paths as a typical user might, and show that PatchCanary does not inadvertently introduce any new bugs. While we do not claim that this test is complete in that it covers all execution paths in the binary, it does provide evidence of the usability of PatchCanary on production software.

5.4 Performance Evaluation

Macro-Performance Tests For our 108 evaluation targets, we measured the compile-time overhead of using PatchCanary. Because PatchCanary is built on Clang to provide source-level transformations, the incurred overhead is confined to the compilation stages. The monitors that we insert are currently simple checks (value, range, type, etc.) and do not impact the runtime of the target program. Runtime would see a performance degradation, however, if we were to insert complex monitors throughout the target program. In the case of PatchCanary, though, we are only inserting monitors which perform passive reporting, and do not alter the control flow of the system. On average, the time for monitor generation and insertion across all of our tests was 712 milliseconds. The distribution of the compile-time overhead measurements are shown in Figure 4.

Runtime Performance Tests For this test, we took each of our codebases and manually triggered each of the CVEs we evaluated. This was done both with the regular patched program, $\mathcal{P}$, and that with augmented patches, $\mathcal{P}'$. We measured the time it took for each respective codebase to handle the vulnerability-triggering input, with the goal of determining the overhead incurred by PatchCanary's monitoring functionality being invoked. On average, PatchCanary imposed an additional 0.75μs of runtime. It is worth noting, however, that this overhead incurred is primarily due to PatchCanary's monitors writing out alerts to log files, which is extra I/O operations. In the case of non-malicious input being passed, there is no extra overhead incurred as PatchCanary's monitors are never triggered.

5.5 Case Studies

We have shown at a high-level how PatchCanary automates the process of monitor generation (see §3). Next, we provide two case studies for selected CVEs to illustrate in more detail how PatchCanary works. Here, we cover an example from an instance where PatchCanary is able to fully generate a monitor, and one where some component of the monitor could not be inferred, thus the monitor could not be generated.

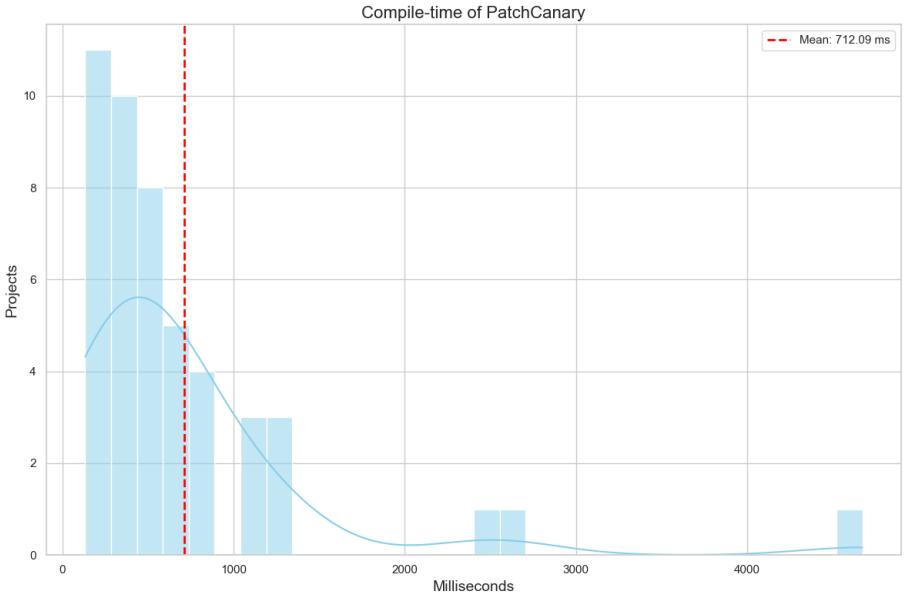

Fig. 4. Time for PatchCanary to generate and insert monitors.

FFmpeg (CVE-2020-12284) For this instance of generating a monitor for CVE-2020-12284, PatchCanary was able to fully automate the process. That is, PatchCanary was able to determine the target variable and its value to monitor for, as well as the constraint on the value that would trigger the vulnerability. The necessary constraints for triggering the vulnerability were inferred from the diff of the ASTs before and after applying the patch. When parsing the respective ASTs, PatchCanary found that the function `cbs_jpeg_split_fragment` had a modification (addition type) that checked the length of a variable. This missing length check from the original code is what caused the heap-based buffer overflow vulnerability outlined in the CVE. PatchCanary was able to automatically generate a monitor that would check the length of the variable, and log an alert if the length was greater than the constraint.

Linux Kernel (CVE-2017-13715) When running PatchCanary on CVE-2017-13715, the monitor generation was semi-automated. That is, PatchCanary was only able to determine where the modifications happened in the codebase, but was unable to infer the necessary constraints for the monitor. The `__skb_flow_dissect` function was properly identified as the target function to monitor, but the modifications to the function were unable to be handled by PatchCanary. The modifications consisted of replacing the statement `return true` with `goto out_bad`, which is a pattern we were unable to generate the appropriate constraints for automatically.

6 Limitations and Future Work

While our approach to monitor generation for patched code introduces significant improvements in detecting potential exploit attempts, there are several limitations that need to be addressed. First, the reliance on symbolic execution and constraint solvers, such as Z3, can introduce performance overhead, especially for large and complex codebases. This overhead can affect both the compilation time and the runtime performance of the instrumented binaries. Second, our current method assumes that the differences between pre-patch and post-patch versions can be effectively captured and encoded as symbolic constraints. However, certain subtle changes or complex logical conditions may not be fully represented, potentially leading to incomplete monitoring coverage.

We envision PatchCanary to be most useful in a community-driven way, much like how YARA signatures work, where signatures are crowdsourced and shared among users [6]. The key insight here is that independent analysts can contribute patches with monitors to our open source project that can then be used by other users in their organizations.

Techniques from the domain of automated program repair can also be used in conjunction with PatchCanary. While orthogonally related, a useful future work may be to use PatchCanary to augment patches that are automatically-generated or synthesized to not only provide monitoring, but to show that the patch is actively mitigating a threat. To the best of our knowledge, there is currently no standard way of measuring the efficacy of security patches. Using PatchCanary, it would be possible to measure every exploit attempt for a specific vulnerability, giving system administrators a measurable indicator that the patches they applied are actively working in their described way.

7 Related Work

In this section, we briefly survey previous related research. We start by looking at previous research in detecting network attacks and lateral movement in particular, and then continue by surveying research on program patching.

7.1 Attack Detection Research

Intrusion Detection. Intrusion detection techniques generally fall into two main categories: *misuse*-based and *anomaly*-based techniques. Misuse-based techniques [11, 33, 39, 45, 52] focus on learning what known attacks look like, and identifying attacks with the same characteristics in the future. Misuse-based techniques range from building signatures of known attacks [45, 52] to leveraging machine learning to identify similar attacks [33]. Anomaly detection techniques [12, 13, 34] focus on learning what the normal activity on a network looks like and flag anomalies as potential attacks. More recently, research in intrusion detection has started focusing on specific types of attacks, developing more

specialized systems, for example to detect web-based attacks [31], botnet infections [22,23], or malicious file downloads [40,43,44]. Compared to previous work in this domain, PatchCanary is different because it monitors inputs targeting patched vulnerabilities.

Alert Correlation. Intrusion detection systems are designed to provide information about a single attack, but modern attacks usually unfold across a number of steps [46]. The field of *alert correlation* [15,51] focuses on analyzing the alerts intrusion detection systems, and provide higher-level information on attempted intrusions. A number of approaches have been proposed to provide effective alert correlation [15, 27, 29, 50, 51].

Lateral Movement Research. Despite the importance of lateral movement attacks, the research in this space is very limited. Ho et al. [25] study lateral phishing, a type of lateral movement in which attackers progressively compromise machines in a corporate setting by sending spearphishing emails from an initial compromised account to further the breach. Fawaz et al. [19] propose a system that builds a graph from the network connection between hosts in a network to detect lateral movement attacks; and, *Latte* [35] focuses on a graph-based representation to model a network for detecting lateral movement attacks. In this paper, we take an alternative approach to this existing research [42,48], and propose the first monitor-based system that can detect an attacker within an organization who is trying to exploit known vulnerabilities as part of a typical lateral movement attempt.

7.2 Program Patching

The challenging task of program patching and modification has been extensively studied [24, 26, 28, 56, 59]. For example, BinSurgeon [20], and AutoFix-E [55] allow users to write patches using templates or source code annotations. Duan, et al. [17] proposed OSSPatcher, which patches vulnerable open source mobile applications with source patches. The work on *honey-patches* [8] presented a way to reformulate security patches in a way that misleads or frustrates potential attackers. Furthermore, techniques in the area of program repair [21, 36, 41, 57] seek to minimize user effort to efficiently fix bugs in software. Other tools work assume that patches are available publicly, or from the analyst [2, 9, 14]. Binary-rewriting [10,18,53,58] and hot-patching at runtime [9,14] are also viable patching techniques; however, we focus on precisely targeting individual functions that can be monitored to catch early indicators of compromise. These techniques are solely concerned with applying a fix to a program, while PatchCanary's focus is on monitoring the functions those patches modified.

8 Conclusion

In this paper, we propose the idea that monitoring even attempted exploits against patched vulnerabilities provides invaluable information for revealing the presence of an attacker in an organization, and we introduce PatchCanary,

a framework for augmenting source patches to monitor for early indicators of compromise. By specifically targeting known-vulnerable functions from CVE disclosures, PatchCanary is able to precisely inject monitors that watch for specific data flows to the functions that were modified in the patch. To allow vulnerability patching to become a powerful contributor to the detection of lateral movement attacks, we propose the novel idea of "patch and monitor" as an alternative to the traditional mindset of "patch and move on." Evaluation on 75 real-world programs shows that PatchCanary is able to automatically augment source patches for 95.9% of the target vulnerable paths to monitor for potentially malicious input while incurring minimal overhead.

References

1. cve-search - a tool to perform local searches for known vulnerabilities (Nov 2021), `https://github.com/cve-search/cve-search`
2. kpatch (Oct 2021), `https://github.com/dynup/kpatch`
3. Shodan (Jan 2021), `https://www.shodan.io/`
4. Libtooling is a library to support writing standalone tools based on clang. (2022), `https://clang.llvm.org/docs/LibTooling.html`
5. The mission of the cve program is to identify, define, and catalog publicly disclosed cybersecurity vulnerabilities. (Oct 2022), `http://cve.mitre.org`
6. Yara - the pattern matching swiss knife for malware researchers (Jun 2022), `http://virustotal.github.io/yara/`
7. Ahmed, M., Mahmood, A.N., Hu, J.: A survey of network anomaly detection techniques. Journal of Network and Computer Applications **60**, 19–31 (2016)
8. Araujo, F., Hamlen, K.W., Biedermann, S., Katzenbeisser, S.: From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In: Proceedings of the 2014 ACM SIGSAC conference on computer and communications security. pp. 942–953 (2014)
9. Arnold, J., Kaashoek, M.F.: Ksplice: Automatic rebootless kernel updates. In: Proceedings of the 4th ACM European conference on Computer systems. pp. 187–198 (2009)
10. Arras, P.A., Andronidis, A., Pina, L., Mituzas, K., Shu, Q., Grumberg, D., Cadar, C.: Sabre: load-time selective binary rewriting. International Journal on Software Tools for Technology Transfer pp. 1–19 (2022)
11. Barbara, D., Wu, N., Jajodia, S.: Detecting novel network intrusions using bayes estimators. In: Proceedings of the 2001 SIAM International Conference on Data Mining. pp. 1–17. SIAM (2001)
12. Bhuyan, M.H., Bhattacharyya, D.K., Kalita, J.K.: Network anomaly detection: methods, systems and tools. Ieee communications surveys & tutorials **16**(1), 303–336 (2013)
13. Chandola, V., Banerjee, A., Kumar, V.: Anomaly detection: A survey. ACM computing surveys (CSUR) **41**(3), 1–58 (2009)
14. Chen, Y., Zhang, Y., Wang, Z., Xia, L., Bao, C., Wei, T.: Adaptive android kernel live patching. In: 26th {USENIX} Security Symposium ({USENIX} Security 17). pp. 1253–1270 (2017)
15. Cuppens, F., Miege, A.: Alert correlation in a cooperative intrusion detection framework. In: IEEE Symposium on Security and Privacy (2002)

16. DeGonia, T.: Cyber kill chain model and framework explained (Mar 2020), https://cybersecurity.att.com/blogs/security-essentials/the-internal-cyber-kill-chain-model
17. Duan, R., Bijlani, A., Ji, Y., Alrawi, O., Xiong, Y., Ike, M., Saltaformaggio, B., Lee, W.: Automating patching of vulnerable open-source software versions in application binaries. In: NDSS (2019)
18. Duck, G.J., Gao, X., Roychoudhury, A.: Binary rewriting without control flow recovery. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 151–163 (2020)
19. Fawaz, A., Bohara, A., Cheh, C., Sanders, W.H.: Lateral movement detection using distributed data fusion. In: 2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS). pp. 21–30. IEEE (2016)
20. Friedman, S.E., Musliner, D.J.: Automatically repairing stripped executables with cfg microsurgery. In: 2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops. pp. 102–107. IEEE (2015)
21. Goues, C.L., Pradel, M., Roychoudhury, A.: Automated program repair. Communications of the ACM **62**(12), 56–65 (2019)
22. Gu, G., Perdisci, R., Zhang, J., Lee, W.: Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection (2008)
23. Gu, G., Porras, P.A., Yegneswaran, V., Fong, M.W., Lee, W.: Bothunter: Detecting malware infection through ids-driven dialog correlation. In: USENIX Security Symposium (2007)
24. Heinricher, A., Williams, R., Klingbeil, A., Jordan, A.: Weldr: fusing binaries for simplified analysis. In: Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis. pp. 25–30 (2021)
25. Ho, G., Cidon, A., Gavish, L., Schweighauser, M., Paxson, V., Savage, S., Voelker, G.M., Wagner, D.: Detecting and characterizing lateral phishing at scale. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 1273–1290 (2019)
26. Huang, Z., Lie, D., Tan, G., Jaeger, T.: Using safety properties to generate vulnerability patches. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 539–554. IEEE (2019)
27. Janakiraman, R., Waldvogel, M., Zhang, Q.: Indra: A peer-to-peer approach to network intrusion detection and prevention. In: WET ICE (2003)
28. Jiang, J., Xiong, Y., Zhang, H., Gao, Q., Chen, X.: Shaping program repair space with existing patches and similar code. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis. pp. 298–309 (2018)
29. Kannadiga, P., Zulkernine, M.: Didma: A distributed intrusion detection system using mobile agents. In: SNPD-SAWN (2005)
30. Kim, B.C., Chen, P.Y., Mukhopadhyay, T.: The effect of liability and patch release on software security: The monopoly case. Production and Operations Management **20**(4), 603–617 (2011)
31. Kruegel, C., Vigna, G.: Anomaly detection of web-based attacks. In: ACM SIGSAC Conference on Computer and Communications Security (CCS) (2003)
32. Lazarevic, A., Ertoz, L., Kumar, V., Ozgur, A., Srivastava, J.: A comparative study of anomaly detection schemes in network intrusion detection. In: Proceedings of the 2003 SIAM international conference on data mining. pp. 25–36. SIAM (2003)
33. Lee, W., Stolfo, S.: Data mining approaches for intrusion detection (1998)
34. Lee, W., Xiang, D.: Information-theoretic measures for anomaly detection. In: Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001. pp. 130–143. IEEE (2000)

35. Liu, Q., Stokes, J.W., Mead, R., Burrell, T., Hellen, I., Lambert, J., Marochko, A., Cui, W.: Latte: Large-scale lateral movement detection. In: MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM). pp. 1–6. IEEE (2018)
36. Long, F., Rinard, M.: Automatic patch generation by learning correct code. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 298–312 (2016)
37. Mitre: Mitre ATT&CK. `https://attack.mitre.org/`
38. Moura, L.d., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
39. Mukherjee, B., Heberlein, L.T., Levitt, K.N.: Network intrusion detection. IEEE network **8**(3), 26–41 (1994)
40. Nachenberg, C., Wilhelm, J., Wright, A., Faloutsos, C.: Polonium: Tera-scale graph mining and inference for malware detection. In: SIAM International Conference on Data Mining (2011)
41. Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: Semfix: Program repair via semantic analysis. In: 2013 35th International Conference on Software Engineering (ICSE). pp. 772–781. IEEE (2013)
42. Noureddine, M.A., Fawaz, A., Sanders, W.H., Başar, T.: A game-theoretic approach to respond to attacker lateral movement. In: International Conference on Decision and Game Theory for Security. pp. 294–313. Springer (2016)
43. Rahbarinia, B., Balduzzi, M., Perdisci, R.: Real-time detection of malware downloads via large-scale url− > file− > machine graph mining. In: ACM ASIA Conference on Computer and Communications Security (ASIACCS) (2016)
44. Rajab, M.A., Ballard, L., Lutz, N., Mavrommatis, P., Provos, N.: Camp: Content-agnostic malware protection. In: ISOC Network and Distributed Systems Security Symposium (NDSS) (2013)
45. Roesch, M., et al.: Snort: Lightweight intrusion detection for networks. In: Lisa. vol. 99, pp. 229–238 (1999)
46. Shen, Y., Stringhini, G.: Attack2vec: Leveraging temporal word embeddings to understand the evolution of cyberattacks. In: USENIX Security Symposium. pp. 905–921 (2019)
47. Steinwart, I., Hush, D., Scovel, C.: A classification framework for anomaly detection. Journal of Machine Learning Research **6**(2) (2005)
48. Tian, Z., Shi, W., Wang, Y., Zhu, C., Du, X., Su, S., Sun, Y., Guizani, N.: Real-time lateral movement detection based on evidence reasoning network for edge computing environment. IEEE Transactions on Industrial Informatics **15**(7), 4285–4294 (2019)
49. Tripwire: The MITRE ATT&CK Framework: Lateral Movement. `https://www.tripwire.com/state-of-security/mitre-framework/the-mitre-attck-framework-lateral-movement/`
50. Valeur, F., Vigna, G., Kruegel, C., Kemmerer, R.A.: Comprehensive approach to intrusion detection alert correlation. IEEE Transactions on dependable and secure computing **1**(3) (2004)
51. Vasilomanolakis, E., Karuppayah, S., Mühlhäuser, M., Fischer, M.: Taxonomy and survey of collaborative intrusion detection. ACM CSUR **47**(4), 55 (2015)
52. Vigna, G., Kemmerer, R.A.: Netstat: A network-based intrusion detection approach. In: Proceedings 14th Annual Computer Security Applications Conference (Cat. No. 98EX217). pp. 25–34. IEEE (1998)
53. Wang, R., Shoshitaishvili, Y., Bianchi, A., Machiry, A., Grosen, J., Grosen, P., Kruegel, C., Vigna, G.: Ramblr: Making reassembly great again. In: NDSS (2017)

54. Wang, S., Wen, M., Chen, L., Yi, X., Mao, X.: How different is it between machine-generated and developer-provided patches?: An empirical study on the correct patches generated by automated program repair techniques. In: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). pp. 1–12. IEEE (2019)
55. Wei, Y., Pei, Y., Furia, C.A., Silva, L.S., Buchholz, S., Meyer, B., Zeller, A.: Automated fixing of programs with contracts. In: Proceedings of the 19th international symposium on Software testing and analysis. pp. 61–72 (2010)
56. Williams, R., Ren, T., De Carli, L., Lu, L., Smith, G.: Guided feature identification and removal for resource-constrained firmware. ACM Transactions on Software Engineering and Methodology (TOSEM) **31**(2), 1–25 (2021)
57. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Transactions on Software Engineering **42**(8), 707–740 (2016)
58. Xie, J., Fu, X., Du, X., Luo, B., Guizani, M.: Autopatchdroid: A framework for patching inter-app vulnerabilities in android application. In: 2017 IEEE International Conference on Communications (ICC). pp. 1–6. IEEE (2017)
59. Zhang, X., Zhang, Y., Li, J., Hu, Y., Li, H., Gu, D.: Embroidery: Patching vulnerable binary code of fragmentized android devices. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 47–57. IEEE (2017)