Multifocal Relational Analysis for Assured Micropatching (MRAM) Final Report

AARNO LABS and MIT CSAIL

Abstract

In the DARPA AMP program, we developed, under the MRAM project, a prototype end-to-end binary patching platform and associated workflow that significantly reduces the costs of patching stripped binaries, the CodeHawk Binary Patcher (CBP). CBP's cost reduction is achieved by lifting the binary into a form that can be edited by a typical software engineer (as opposed to the rare and highly skilled reverse engineer). Concomitantly, our platform drastically decreases the likelihood of errors by formally validating translation steps of the process and by providing the developer with assurance artifacts for review. Furthermore, the platform enacts minimally invasive changes to the binary, with the ability to reason about individual expressions and instructions, maintaining the testing and certifications applied to the original binary. We have implemented a prototype binary patching and assurance graphical user-interface workflow as a plugin to the binary analysis platform Binary Ninja. The MRAM project was employed and evaluated by independent operators from the AMP TA3 performer, with the experience being described as "smooth," "thorough," and "intuitive."

1 Overview

Most organizations do not consider binary patching a viable solution. They are left with no options (other than workarounds) for fixing known issues on systems with incomplete source code or lost build environments. Understanding and patching a stripped binary is a challenging task. Raw binary code (assembly code) is not meant for human consumption and modification. Currently, binary patching is an expensive process that involves great effort to (1) understand the binary instructions and form a high-level understanding of a flaw, (2) formulate a patch on those instructions that achieve the desired changes, (3) understand the repercussions of the low-level changes in the overall binary, and (4) test those changes in a best effort fashion. Reverse engineers are rare and expensive, and the binary patching process is ad-hoc, missing trustworthy tooling, and lacking formal validation and verification of results.

We have developed the CodeHawk Binary Patching (CBP) platform to democratize binary patching, decreasing costs and increasing assurance.

1.1 Contributions

Transition-Ready, Independently-Evaluated Binary Patching Workflow. We have developed an effective and efficient binary patching workflow in the context of our BinaryNinja UI plugin. The workflows enable a developer to stay completely within the UI to patch a stripped binary, leading the developer through patching the C lifting, automatically applying the patch, reviewing the assurance artifacts, and additional patch iterations (if required). This complete end-to-end workflow, including matching source and automatically introducing source symbols and types, was independently demonstrated to be revolutionary during AMP's Phase II end evaluation: we achieved the fastest time to patch, coupled with the smallest patch changeset produced. An independent team of 2 developers was able to create a correct micro-patch on a stripped binary in 20 minutes (without live guidance), the fastest time to patch of any team participating in the evaluation. The team also independently reviewed the assurance result and concluded the patch was correct. The resulting patch had a smaller changeset versus a patch created by reverse engineers on the evaluation team (10 bytes vs 57 bytes changed). This demonstrated our prototype is ready to be used by transition partners for rapid and assured binary micro-patching.

Binary Patching Without Reverse-Engineering. The CodeHawk Binary Patching system enables an analyst with only source-code development experience to patch binaries without running nor recompiling the original binary. CBP performs a validatable lifting of the binary to editable C-source code with symbols and types incorporated

Authors' Contact Information: Aarno Labs, mgordon@aarno-labs.com; MIT CSAIL, rinard@csail.mit.edu.

from the Binary Ninja type recovery system, including BSI source matching. The source code lifted by CBP looks similar to the original source code in many cases and employs natural expressions on the source types. It is straightforward for an analyst to understand the lifted code to determine how to modify the lifting to enact the patch. Then, once the analyst has enacted the patch, CBP automatically applies the changes on the binary, as defined by the well-defined semantic changes on the lifting. No recompilation is required. CBP reasons about the associations of the lifted source to the binary to automatically generate binary instructions to access values and variables efficiently, keeping changed code lean and fast. The analyst can next review the assurance artifacts. All these steps are performed in our user-friendly Binary Ninja plugin. Finally, the analyst can test the modified functionality.

Intuitive Patch Assurance Workflow for Typical Developer. Enacting a binary patch is straightforward in CBP, just modify source code. The next step is for the analyst to decide if the patched binary is correct, i.e., assuring the modifications to the binary. The CBP patch assurance artifacts and workflow do not require a PhD in program analysis to comprehend and correctly employ. After much experimentation and feedback from independent analysts on AMP, we have developed an effective analyst workflow where CBP helps to assure that (1) the patch produced on the binary correctly represents the changes enacted on the lifting and (2) the patched lifting correctly patches the vulnerability it is meant to fix. For (1), we provide assurance artifacts in our UI that enable the analyst to rapidly understand what the patch has changed in the binary (functions, control flow, etc.) via relational analyses and then, once the change has been localized, to directly compare the modified lifting (produced by the developer) with the lifting of the *patched* binary (produced by our validated translation from binary to source). We have demonstrated this capability in the AMP Phase II end evaluation, with the evaluating analysts stating that "the creation of custom patches/verification was intuitive." For (2), we employ CodeHawk's C Analyzer on the patched lifting to demonstrate whether the vulnerability has been fixed; this applies to the large class of memory-corruption vulnerabilities.

Automatic Source Symbol and Type Incorporation. A vital synergy exists between BSI and CodeHawk. BSI is STR's TA1 performer in AMP. BSI will find source matches for functions in the binary. For high-confidence matches, the symbols and types of the source code will be automatically incorporated into CodeHawk's lifting and analysis. This capability was demonstrated during AMP's Phase II End evaluation.

Minimally-Invasive Binary Patching. When CBP performs the modifications to the original binary specified by the changes on the lifting, CBP reasons about the required changes at the expression level by comparing the abstract syntax tree (AST) of the original lifting to the modified lifting. Thus, CBP knows what has changed at the expression level and how to enact those changes on the binary. The changes it enacts are minimally invasive, meaning as few bytes are modified as required by the semantics of the changes on the lifting. Minimal changes enable precise relational analysis and have the best chance to preserve the vast majority of certification and testing that went into the original binary.

Translation Validation. An essential assurance component is the validation of the translations employed in the binary patching workflow. The analyst must be able to trust the results of a tool, and thus, a tool must demonstrate it is trustworthy. CodeHawk is currently able to create checkable proofs for its analysis results. For the AMP program, we achieved foundational steps towards a patching validation infrastructure (PVI) whose goal is to validate the translation steps of CodeHawk. When complete, PVI will produce checkable proofs demonstrating the correctness of translations between 1) binary to lifted source and 2) lifted source to binary. This is a massive leap in assurance over unverified tools that translate in a best effort approach, even when their input seems well-defined, without validation, how can one be sure a tool is bug-free?

Fully-Automated Discovery and Patching of Vulnerabilities. CodeHawk includes vulnerability discovery and patch templates that automatically find and fix important classes of vulnerabilities. Currently, CodeHawk supports string overflow vulnerabilities that occur through insecure use of C library functions.

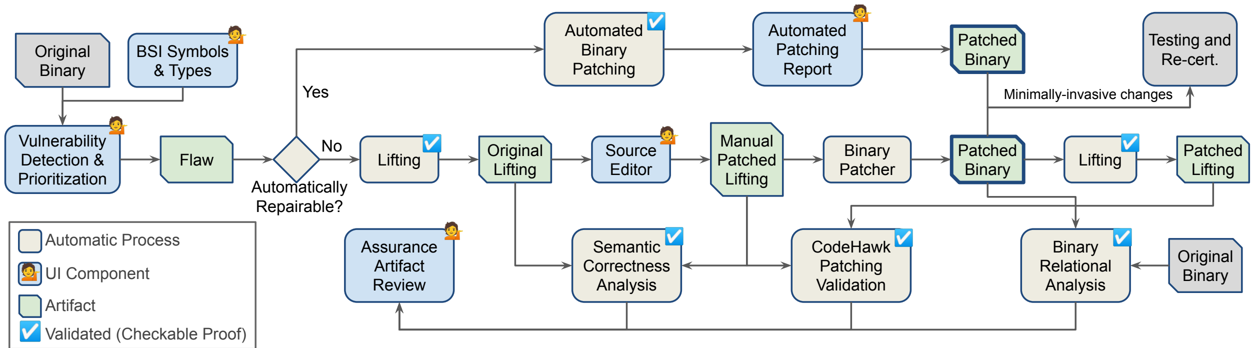


Fig. 1. CodeHawk Binary Patcher System Diagram

1.2 System Design

Figure 1 provides an overview of the CodeHawk Binary Patcher system. The system begins by automatically discovering and prioritizing vulnerabilities on the stripped binary after the binary has been enriched with source type and symbol information by the BSI source correlations. This is achieved by automatically searching for vulnerability templates or manually consulting the output of CodeHawk's C Analyzer to find potential sources of vulnerabilities on the lifting.

Next, an analyst chooses a vulnerability or vulnerability class to fix by either automated patching or human-in-the-loop patching.

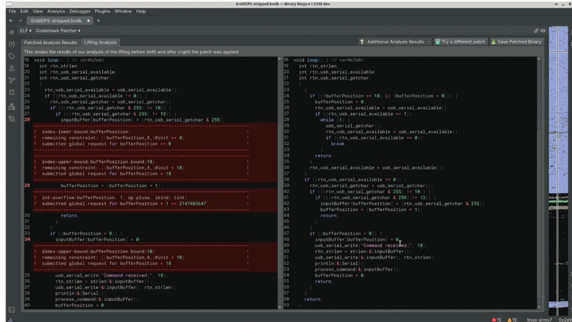
In automated patching, our system repairs the binary automatically and provides assurance evidence to the analyst. This can be done for many important vulnerability classes manifested by undefined behaviors in the original C source code. We then have a disciplined and checkable proof of transformation for each patched vulnerability instance. The transformation is fully automated, even finding or making free space on the original binary.

In developer-guided patching, a developer modifies a high-level C lifting of the function(s) required to fix the flaw. The high-level lifting includes idiomatic source symbols and type usage enabled by BSI's source correlations. After producing a patch on the C lifting, the system will automatically produce the associated changes on the binary. Next, the analyst will consult automatically-produced assurance artifacts to decide if the patch fixes the flaw (Semantic Correctness Analysis) and whether the patch was correctly applied to the binary (CodeHawk Patching Validation and Binary Relational Analysis). All the translation and analysis steps in the process are designed to produce checkable proofs that can be used to validate each step. These checkable proofs provide strong evidence that the step in question is correct.

Figure 2 provides examples of two of the important steps in our workflow. In Figure 2a, the user is shown, on the left, the issues discovered by CodeHawk's C-Analyzer that could lead to potential vulnerabilities in the lifting of the original binary. On the right, it is demonstrated that no issues are found on the patched lifting. This indicates that the patch has fixed the vulnerability. Figure 2b provides an example of the "Patch Summary" screen, which provides details of the changes made to the binary and the validations performed on the translations and transformations. An analyst can review this screen to understand the changes made rapidly and whether they can be trusted.

2 Background

The CodeHawk Binary Analyzer is part of the CodeHawk Analyzer Tool Suite, a suite of analyzers based on the mathematical theory of abstract interpretation [9, 10]. The CodeHawk Analyzer Tool Suite has been in development since 2008, originally by Kestrel Technology LLC, and since 2020 by Aarno Labs LLC. Its development has been funded by grants from DARPA, IARPA, DHS, and AFRL, and by internal research funds. The Tool Suite was made open source in 2019 and is available under MIT License on GitHub [1].


(a) C-Analyzer showing the potential issues with the original lifting (on left in red), and no issues found on patched lifting.

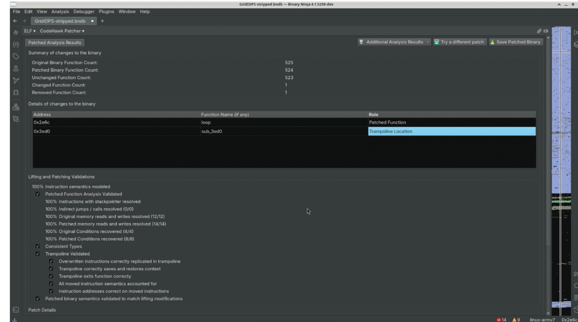

(b) Patching and lifting overview page provides details of validations and changes to binary.

Fig. 2. Examples of screens in the CBP Binary Ninja Plugin.

At present, the CodeHawk Analyzer Tool Suite consists of

- A language-independent abstract interpretation engine;
- A sound C-source code analyzer front end focused on proving the absence of undefined behavior;
- A binary analyzer front end targeted at variable discovery, deep data flow analysis, and memory safety analysis

2.1 Abstract Interpretation Engine

The CodeHawk Abstract Interpretation Engine is a language-independent tool to generate invariants. It has its own internal representation language, called CHIF (CodeHawk Internal Form), an imperative register-based language that supports primitive and structured types, structured and hierarchical control flow, basic arithmetic operations, basic set operations, and linear and set-theoretic predicates. Abstract interpretation is performed using numerical and symbolic abstract domains. The current numerical domains include intervals, linear equalities, and linear inequalities (polyhedra). Symbolic domains currently include taint representation and symbolic sets. In addition to these domains, the abstract interpretation engine provides the Value Set domain [3, 4], a combination of numerical offsets and symbolic base values, which provides constrained relational analysis at the computational cost of non-relational analysis, which is especially useful in the analysis of executable code.

The heart of the abstract interpretation engine is a set of extremely efficient iterators that perform the value propagation and widening and narrowing over the various abstract domains.

2.2 Abstract Domains

An abstract domain is a decidable theory. It consists of a finite or infinite set of elements that form a lattice, with well-defined meet and join operations and a bottom and top element. Furthermore, it provides forward and backward transformers for all dataflow constructs in the language as well as for assert statements. The transformers are guaranteed to be an over-approximation of the concrete semantics of the operations modeled. Custom domains may include an arbitrary number of domain operations that define transformers for custom constraint generation. The core system provides several numerical and symbolic domains, including intervals, linear equalities and linear inequalities, and symbolic sets, as well as domains specialized for binary analysis, including the strided interval domain and the value-set domain.

2.3 CodeHawk Binary Analyzer

The CodeHawk Binary Analyzer uses abstract interpretation to recover source-level constructs such as local variables, structure formats, function signatures, types, and meaningful branch predicates. It has its own disassembler, which originally targeted x86 and MIPS, and for AMP, we added support for ARM32 and Thumb1/2. Other targets

size	26.6 MB
instructions	3,963,865
functions	59,170

stage	time (secs)
disassembly	18
function construction	433
analysis (12 rounds)	49,229
file handling (I/O)	$\sim 8,800$

function coverage	98.3%
stackpointer precision	99.5%
reads precision	86.3%
writes precision	90.7%

Table 1. Analysis of an x86 linux kernel binary

can be added in a modular fashion. Having an integrated disassembler allows direct feedback of analysis results into the disassembly to resolve, for example, indirect jumps. CodeHawk comes with a large collection of library function summaries (currently more than 2500), including many summaries for the standard C library and dozens of Microsoft libraries. These summaries provide function signatures, preconditions on arguments, postconditions on return values, and side effects on arguments. New summaries can be easily added, e.g., for VXWorks, Integrity RTOS, etc.

2.3.1 Disassembler and Function Construction. The CodeHawk Binary Analyzer has its own disassembler. Initial disassembly is performed in Linear Sweep fashion to obtain maximum coverage of the code section. One disadvantage of Linear Sweep is the need to deal with data embedded within the code section. It is our experience that for the vast majority of regular executables encountered, our Linear Sweep approach combined with embedded data recognition provides perfect disassembly. For the rare cases that are not handled correctly, global analysis can contribute to locating code boundaries.

Function entry points are initially identified by application entry points and direct call targets. Function entry points can also be imported from other tools. For each function entry address, a function is constructed by recursive descent. Indirect jumps are resolved, if possible, against the jump tables identified in the disassembly phase. Non-returning calls are identified and used to terminate branches.

2.3.2 Scalability. Deep data-flow analysis takes time. This makes the CodeHawk Binary Analyzer essentially a batch tool. The tool comes with an extensive command-line interpreter that allows configuring analyses to be performed on multiple binaries in parallel. In the past, we have applied CodeHawk to thousands of malware binaries for advanced feature extraction. In AMP, we applied CodeHawk to HTTP daemons from 400 different routers, more than 200MB in total, to identify vulnerabilities (uncovering more than 100,000 potential string buffer overflows in the process). CodeHawk also scales to large individual binaries; Table 1 shows some timing and precision results from analyzing an x86 Linux kernel.

All analysis results are saved (in highly compressed form) to disk, accessible via a comprehensive, programmable Python API, for interactive querying, further analysis, and correlation, or for import in other tools. This capability allows the CodeHawk Binary Analyzer to be easily integrated with other tools to navigate the results and apply the results to other tasks, like lifting and patching. We have done this integration succesfully in AMP, creating deep integrations with BinaryNinja and IDAPro.

2.4 CodeHawk C Analyzer

Like the CodeHawk Binary Analyzer, the CodeHawk C Analyzer is built on top of the Abstract Interpretation Engine. It takes as input a single C file or an entire C project with a Makefile; it uses CIL [17], as a front end to parse the pre-processed source code into OCaml data structures, which form the basis for the analysis.

The objective of the current C Analyzer is to prove the absence of undefined behavior. It does so by automatically generating proof obligations for every construct in the program that can possibly lead to undefined behavior (on average 2 - 5 proof obligations per line of code), and uses invariants generated by the abstract interpretation engine to discharge these proof obligations. Depending on the quality of the code the percentage of proof obligations thus automatically discharged can vary from 65% to 95%. For the remaining, undischarged proof obligations, detailed information is presented to the developer which part of the proof obligation was unsupported by the invariants, which can either prompt the developer to modify the code (if those conditions are recognized as

potential errors), or to provide intermediate assertions to strengthen the invariants (which themselves are subject to checking their validity by propagation). All invariants applied to discharge the proof obligations are available for independent checking and may serve as evidence for auditing.

We have applied the CodeHawk C Analyzer to more than 1.6M lines of code, including networking applications: openssl, nginx, wpa_supplicant, dnmasq, and lighttpd; desktop applications: cairo, dovecot, and git; and embedded applications: cleanflight (drone navigation) [12].

3 CodeHawk: Binary Analysis and Lifting

The CodeHawk Binary Patching (CBP) platform consists of a suite of binary analyses that extract detailed information and semantics from a binary, and lift those semantics to a high-level editable format: parseable C code. Using these translations and analyses, we have created the CBP developer patching workflow. This workflow enables a typical developer to understand a flaw on a stripped binary, and then enact the binary patch as modifications to C code. Next, the workflow enables a typical developer (not a PhD in program analysis) to decide if (1) the patch correctly fixes the flaw (providing a yes/no answer for certain important types of vulnerabilities) and (2) the patch was correctly applied to the binary.

When source symbols (e.g., names and types) can be correlated with the binary, the lifting incorporates these symbols to create intuitive high-level C code with structures accessed as they would be in human-developed code. For the AMP program, we developed a capability for CodeHawk to ingest the symbol associations found by BSI. During AMP, we worked on producing easy-to-read liftings and demonstrated this for many functions from stripped production binaries.

CBP is offered to developers via Binary Ninja user interface plugins. The CBP workflow takes over after an analyst has identified the issue to fix, and has applied BSI to find corresponding source code and associated source symbols and types. Next, the developer can make changes directly on the lifted C source code. When finished with changes, with a single button press, the changes represented on the source are automatically applied to the binary. No manual extraction of information from the binary is necessary in the majority of cases. No recompilation of the binary is required.

After the binary patch has been produced, CBP produces intuitive assurance analysis results grouped into (1) binary relational analysis, and (2) semantic correctness of the patch. Binary relational analysis artifacts enable the developer to decide if the patch was applied correctly to the binary and to understand the impact of the patch. We have displays and reports in our plugin that help the developer. For example, we show a diff view of the C code that was modified by the developer (to represent the patch) and the C lifting of the patched binary that was automatically produced by CBP. A quick glance at this diff view shows if there are any differences between these two C codes. When the semantics are equal, a developer knows that the patch was applied correctly by CBP.

For the semantic correctness of the patch, we employ the CodeHawk C Analyzer to prove the absence of undefined behavior on the modified semantics of the patch. We have demonstrated this during AMP challenges and Hackathons. For example, in Challenge 3, we demonstrated that a patch correctly fixed the underlying vulnerability, and in Target 4 of the Fall 2024 Hackathon, we demonstrated our patch fixed all of the undefined behaviors reported by the C Analyzer in the vulnerable function. Furthermore, in the July 2023 AMP Hackathon, we used our semantic analysis to prove that a binary patch produced by the evaluation team was incorrect and that our automatically produced patch was correct.

3.1 CBP Automated Patching

CBP also allows finding and patching important classes of vulnerabilities automatically. The CodeHawk analysis generates precise invariants for every instruction in the binary. These invariants enable the specification of templates to discover vulnerabilities automatically, and, importantly, they also enable us to specify how we can automatically fix important classes of vulnerabilities. For example, CodeHawk can find instances where it is possible for a library call to overwrite a buffer (because a bound was not correctly specified for a write to the buffer). CBP can then convert the library call to a version where a bound is provided and applied. Alternatively,

CBP can add the appropriate control flow checks to throw an error when an access of or write to a buffer exceeds the computed bounds of an array or buffer.

We have demonstrated this capability in AMP by identifying and fixing string buffer overflow vulnerabilities associated with C-library calls that do not provide bounds. CBP can automatically calculate buffer bounds and convert an unsafe (unbounded call) to a safe call bounded by the calculated buffer (or array) size. We have applied this transformation on over 400 commercial router firmware from 33 different models from ten vendors over two architectures (ARM32, MIPS).

3.2 Lifting and PatchIR Generation

Many binary analysis tools offer some form of decompilation (lifting) of binary code into C source code with the main aim of making the binary code easier to understand. The resulting C source code is often not parseable C code, and, more importantly, the precise relationship between C source code and the original binary code (called provenance) is imprecise and not well defined. The latter is especially problematic in a patching context, where an edit to the lifted source code must be translated into a change in exactly the correct instruction(s) or part(s) of an instruction. To achieve this level of precision in provenance, we developed PIR: Patching Intermediate Representation [19]. PIR aims to provide an intermediate representation that combines a high-level C-like representation with assembly-level data with a well-defined semantics firmly based on the C semantics. The basic representation is an abstract syntax tree (AST) with nodes representing control flow structures, instructions, expressions, and types. The nodes are modeled after the data structures from CIL (C Intermediate Language) [17], a representation that covers the entire C language and has been a popular base representation for many analysis tools for more than 20 years.

The base representation artifacts are the AST, a local symbol table per function, providing types and possibly storage locations, a global symbol table, providing types and storage locations, and so-called *span tables* relating semantic instructions and control flow constructs to ranges of instruction addresses in the executable. Typically, there are multiple levels of ASTs where the highest level is closest to the source code (typically typed, without reference to registers), and the lowest level is close to the assembly code such that each assembly instruction is represented explicitly with base address/offset registers. The links between the layers include (1) the primary connections: linking high-level instructions to low-level instructions (possibly a many-to-many relationship), and high-level lvalues and expressions to low-level lvalues and expressions resp.; (2) reaching definitions: linking variable uses (both registers and memory variables) with the instructions where these variables are defined (that is, their value is set), and linking flag use (in branch and instruction predicates) to the instructions that set those flags; and (3) definitions used: linking variable definitions to the instructions where their value is used, again both for registers and memory variables. The latter two, reaching definitions and definitions used, are important aids to the patcher to determine where a change must be made and what the impact of that change is on the rest of the code in the function (and potentially outside the function). In addition to these required elements of PIR, optional *available expressions* may be included in the representation that inform the patcher if a target expression, identified as the change, is already available in a register or on the stack at the place of use, such that it can be directly reused rather than having to be retrieved and constructed.

The PIR interface is a standalone library implemented in python [19]. It provides constructors for all semantic constructs, a pretty printer that prints parseable C code, a serializer to produce JSON serialization of the entire representation for easy sharing, a deserializer to reconstruct the AST and associated elements, and several visitor classes to allow customization for a variety of transformations by third-party tools.

Construction of PIR for the ARM architecture is fully supported by both the CodeHawk Binary Analyzer and Binary Ninja.

4 Binary Patching

CodeHawk's Binary Patcher supports both fully automated and human-in-the-loop editing modalities. In the fully automated case, the patcher consumes a declarative file format specifying patches to apply to a binary. This format is easily generated by modular tooling for flexible bulk patching. For developer patches, the patcher takes

user edits to CodeHawk's lifted code and infers the required binary patch specification—then follows through. It finds (or makes) space in the binary, transforms and compiles the user's edited code, and modifies the binary to alter its semantics in line with the supplied declarative patch specification file.

The core process takes two abstract syntax trees of pre- and post-edited code and recursively compares nodes to find differences corresponding to user edits. While the literature has explored algorithms for computing AST differences [11, 15, 26], the patcher is not seeking a strictly minimal edit sequence. Instead, the patcher must recognize edit patterns in context. By examining each difference site, the patcher then generates edit operations, such as inserting a new function, altering a subscript index, or deleting a function call.

A key element of patch specification inference, edit operations allow the patcher to produce more finely targeted micropatches in some situations. For example, whereas deleting a function call can be done by overwriting the call instruction with a nop, deleting a loop does not require replacing every statement in the loop body with a nop. Instead, the patcher can insert an unconditional branch to skip past the loop. Other kinds of edits require even more care from the patcher; for example, altering a declared variable's type may require recompiling all uses of that variable. Currently, the patcher supports single-statement insertions and deletions, both atomic statements (function calls and assignments) and compound statements (blocks, loops, conditionals). The patcher also supports mixed insertions and deletions. The patcher also supports many kinds of subexpression replacement patches, including insertion of new strings and replacement of function call arguments. Over the course of AMP, we have built up patch edit operations to support over 95% of patch edits in the AMP Challenge Problems (see Section 7).

When identifying user edits, the patcher can access the rich metadata embedded in the original lifting's PIR to determine not just what edit was made, but what instructions within the binary it affects. For example, a lifted function call identifies the binary-level call instruction, while also connecting the call instruction to the instructions which compute the function arguments that are implicitly passed in registers. This allows the patcher to treat a tweak of an argument to a function call as a distinct edit from a replacement of the whole function call. In doing so the patcher produces simpler and more targeted patches.

Very simple patches (such as altering a numerical constant) can usually be done in-place. More sophisticated patches, especially those that insert new code or new string constants, require the patcher to find or create new space in the binary. The patcher uses a small suite of approaches to accomplish this. In the simplest cases, existing blocks of NOPs used for padding or alignment purposes (and known to not be executed thanks to CodeHawk's extensive binary analysis) can be reused. In many cases, space can also be scavenged from non-NOP padding placed between executable segments. When such measures fail, the patcher can, in many cases, fall back on more invasive surgical modifications to the binary at hand, altering ELF headers to transparently allow the addition of new segments to the binary.

Inserting new code into a compiled binary function involves the use of a *trampoline* to divert control flow and have the user's code execute in the intended manner. These trampolines are platform-specific pieces of assembly code that mediate between the original execution environment and the environment needed to execute the user's code. Trampolines work hand-in-hand with source code transformations to account for details like explicit register manipulations and non-local control flow transfers.

The patcher currently supports 32-bit ARM and Thumb-2 instruction sets, along with 32-bit MIPS. An initial prototype of support for Power32 was in progress at the end of AMP. The bulk of patcher-side development for a new architecture is encapsulated within two files. Proof-of-concept support for a new architecture can be completed in a day or two. Tracking down all the edge cases and hidden assumptions that can be exposed with a new architectural target can take longer, but it contributes to a smooth end-user experience.

5 Patch Assurance

Providing assurance for patches is an important and significant part of the DARPA AMP project. A major challenge in this context has always been to provide a specification for the patch that serves as the basis for verification. We address this problem by separating the assurance of the patch from the question of whether the patch achieves the (often unspecified) intent of the developer. To do so we are developing a framework called *Patching Validation*

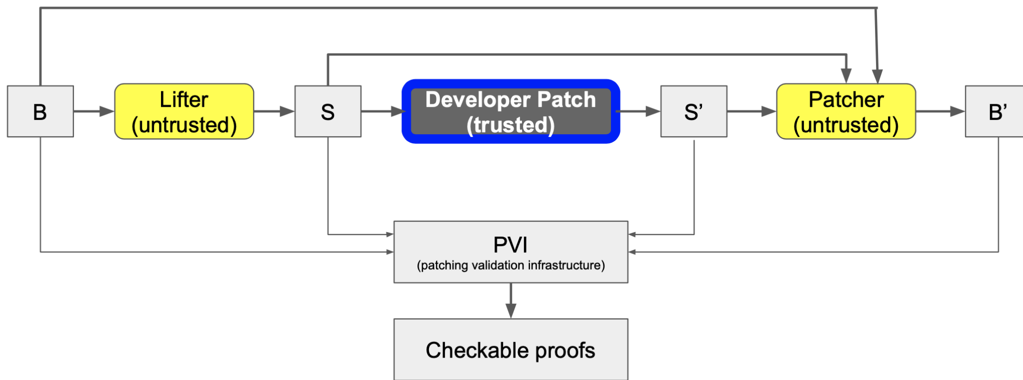


Fig. 3. Overview of Patching Validation Infrastructure

Infrastructure described below, followed by a brief description of the components of this framework that have been developed and demonstrated so far in the AMP hackathons.

5.1 Patching Validation Infrastructure

PVI (Patching Validation Infrastructure) is a generalization of TVI (Translation Validation Infrastructure) [16, 20, 22], developed in the late nineties.

Figure 3 shows an overview of the components involved in PVI and their trust status. Starting with a vulnerable binary B the CodeHawk Lifter produces a lifting in C source code of the function(s) to be patched, S. The developer edits this function (or multiple functions), resulting in function(s) S'. The CodeHawk Binary Patcher determines the semantic differences between S and S' and based on the resulting patch description file (as described before) applies the patch to the vulnerable binary B, producing patched binary B'.

Within this framework it is assumed that the change made by the developer is the intended change and the task of *PVI* is to demonstrate that the semantic change represented by the difference between S and S' is indeed reflected faithfully in the change from B to B', where neither the lifter nor the patcher is trusted, and that B' is a valid binary, or more formally:

(1) **Lifting:** S is a semantically correct representation of B; that is, there exists a simulation relation between the behaviors of B and S. This correctness criterion can be reduced to a set of proof obligations that validate each transition in the behaviors, relying on information provided by the lifter. It is important to note that this information is not assumed to be true, but it simplifies the construction of the proofs since checks can be performed at a higher level of granularity. The approach is similar to (albeit the reverse of) translation validation.
(2) **Patching:** Transformation (B, B') is a correct realization of transformation (S, S'). Again the correctness criterion can be reduced to proof obligations on the transformation simulation relation between (S, S') and (B, B'). Like before the checks can be made more efficient by incorporating information provided by the patcher.
(3) **Binary Integrity** In some cases, the patcher has to allocate extra space in the binary or overwrite existing sections of the binary to achieve its goal. This part of the verification is mostly unrelated to the semantic changes but concerns the validity of the produced binary with respect to adherence to file format specification (e.g., ELF) and possibly related to the architecture (e.g., use of instructions not provided by the target processor).

The goal is to produce proofs (evidence of validity) for all of these steps that can be checked independently of the tool (that is, PVI itself does not need to be trusted either), which, especially in the context of medical devices, is a significant benefit for potential certification, regulation, and auditing requirements.

This approach provides a clean separation of concerns between verifying the patch itself (as provided by the developer in source code) and determining if the patch achieves the original intent, which could include a wide variety of objectives, ranging from fixing a memory safety bug to strengthening encryption to resolving a potential denial-of-service issue, each of which requires very different assessments. These assessments, however, can now be performed at the source level, thus significantly increasing the number of tools that can be deployed, including traditional code reviews, and bug finders.

5.2 Patch Semantic Analysis

As mentioned above, PVI verifies that the patch, as represented in source code, is correctly applied to the binary. This leaves the complementary task to establish that the patch, as entered by the developer, itself achieves its intended objective and does not alter unrelated behavior. Of course, traditional tools such as manual code reviews and testing are available. However, if appropriate, e.g., when fixing a memory safety bug, it would be desirable to produce evidence in the form of proofs of the absence of undefined behavior automatically by a tool like the CodeHawk C Analyzer, as described above in Section 2.4. The CodeHawk Binary Analyzer can produce high-fidelity liftings (validated by PVI) for individual functions, and we apply the C Analyzer to the function being modified by the analyst to demonstrate how the current modifications to the lifting affect the open proof obligations found on the function. We also provide C Analyzer output on the lifting of the modified binary as part of the post-modification assurance artifacts; this demonstrates that the modifications applied (automatically) to the binary by CBP have closed the vulnerability.

5.3 Binary Relational Analysis

Our main assurance efforts so far have focused on providing the developer with sufficient evidence to convince a developer that the patch was applied as expected without any adverse side effects. To achieve this the following artifacts are presented:

- **Summary of Changes** showing a comprehensive list of the functions semantically changed by the patch, including functions that may not have been syntactically changed, but whose semantics changed because, for example, a change in a global variable.
- **Detailed Changes** showing an instruction by instruction report that prints the original and new syntax and semantics of the instructions changed (again either syntactically or semantically) within a changed function side by side.
- **Control Flow Graph Comparison** highlighting the structural differences between the original and patched function control flow graphs, e.g., the inclusion of a trampoline, or the addition or removal of an edge or branch condition.

These artifacts provide the building blocks for the PVI framework described above.

6 A Unified Algebraic Framework of Program Analyses

6.1 Overview

Program analysis is a classic field in computer science. Initially developed for optimizing compilers [2], program analysis has since found wide applications in areas including computer security [23], program verification [25], error detection [8], software engineering, and human-computer interaction.

Despite this history, program analysis today presents a fragmented landscape characterized by point solutions — analyses developed to solve specific problems, with no unifying theory that would enable us to obtain a more comprehensive understanding of the space of program analyses or the relationships between them. Such a theory would help us navigate the space of program analyses to discover previously overlooked analyses, choose analyses to satisfy specific analysis goals, and systematically combine analyses within a clean algebraic reasoning framework.

6.2 An Algebraic Framework for Program Analyses

In AMP, we developed and published [21] a new, unified framework of program analyses. In this framework, each analysis is characterized by a set of program execution traces, with each set of traces corresponding to a program property that the analysis extracts. This property is often made explicit in a *trace predicate* in the form of a logical formula that, given a candidate trace t, returns true if the trace is in the set of traces for the analysis and false otherwise.

Traces provide a unified algebraic framework for comparing and combining both sound and unsound program analyses. Sets of traces ordered by reverse subset inclusion ($\supseteq$) comprise a lattice with least upper bound set intersection ($\cap$) and greatest lower bound set union ($\cup$). This underlying lattice induces a corresponding lattice over program analyses. The lattice order immediately delivers a universal mechanism for comparing analyses, with less precise analyses including more traces than more precise analyses.

Viewing analyses as sets of traces also makes it possible to productively integrate constructs not previously perceived as program analyses into the framework. For example, a set of traces generated by a program fuzzer can be seen as an analysis. Many program analyses assume that certain kinds of errors (for example, out of bounds accesses or accesses via null pointers) do not occur. It is possible to identify these assumptions as sets of traces and integrate them into the framework.

In our work, we primarily consider two kinds of analyses: 1) sound dataflow analyses, which overapproximate the set of program traces, and 2) techniques such as fuzzing and concolic execution, that explore the set of program traces and which we view as unsound analyses that underapproximate the set of program traces. We illustrate how to specify monolithic dataflow analyses that extract multiple kinds of analyses as the conjunction (least upper bound) of multiple sound microanalyses. We also consider unsound analyses that disjunctively combine traces from multiple fuzzing or concolic executions. These unsound analyses can be seen as generalizing the information from the explored executions.

6.3 Sound and Unsound Analyses

Intuitively, a sound analysis produces an analysis result that holds for all possible executions of the analyzed program. More precisely, an analysis is *sound* for a program P if its set of traces includes all of the traces that the program may exhibit when it executes. Soundness proofs for analyses with sets of traces defined by trace predicates typically involve proving that the trace predicate includes all possible program traces. Prominent examples of sound analyses include standard dataflow analyses and exhaustive testing to enumerate all traces.

Intuitively, an unsound analysis produces an analysis result that may not hold for some possible executions of the analyzed program. Examples of unsound analyses include fuzzing, concolic execution, and bounded model checking. Examples of constructs that can be interpreted as unsound analyses include assertions and execution integrity assumptions such as the absence of memory errors, null pointer dereference errors, or API misuse errors.

We note that the framework cleanly highlights the tension between soundness and precision — soundness requires the analysis to include at least all of the traces of the analyzed program, driving the analysis to include more traces, while precision drives the analysis to include fewer traces.

The tension between soundness and unsoundness can sometimes be productively addressed by program transformations that systematically transform the program to eliminate undesirable traces not included in the unsound analysis. Examples include failure-oblivious computing and recovery shepherding.

The least upper bound of a sound analysis and an unsound analysis is a *conditionally sound analysis* — an analysis that is sound given the property specified by the unsound analysis. This approach can cleanly and precisely specify, within a unified framework, the assumptions under which an analysis provides a sound characterization of the potential behavior of the program.

6.4 Program Analysis and Secure Input Parsing

Input parsing code is an important source of security vulnerabilities [14]. One common practice is interleaving parsing and validation code with input processing code (shotgun parsing) [6]. A key problem with this common

practice is that (potentially adversarial) inputs that trigger program errors or target vulnerabilities should be rejected without processing [6, 24]. But starting input processing before input parsing and validation completes can make it difficult to impossible to reject such inputs before processing — by the time the program determines that it should reject the input, it has already been at least partially processed.

One way to attack this problem is to process each input as a transaction, with the input processing effects taking place only when the entire input has been successfully processed [24]. Another approach is to structure the program so that input parsing and validation completes before input processing starts [5]. While domain-specific languages for specifying parsers [18] and parser combinator libraries exist that support this coding practice [5], many developers prefer to manually code their input parsers. In any case, it may be beneficial to structure the computation in three phases as follows. A key aspect of this structure is that if the input parsing or validation phases reject the input, the program behaves as if the input never existed:

- **Input Parsing Phase:** This phase reads the input and populates the data structures that hold the various parts of the input. It rejects inputs that are not syntactically correct.
- **Input Validation Phase:** This phase reads the data structures that hold the parsed data to check that it is possible to process the input fully. If not, this phase rejects the input. It may also store relevant input data into different data structures that the input processing phase reads.
- **Input Processing Phase:** This phase processes the parsed and validated input. It is the only phase that performs any effects that persist beyond the computation that the input triggers.

Each of these three phases is characterized by specific computational patterns that can be verified by appropriate program analyses targeting each phase's code. Analyses for the input parsing phase, for example, may verify that the parsing code writes only data structures that are designed to hold input data. If the input format can be specified by a deterministic context-free grammar or regular grammar, the analysis may verify that the structure of the code corresponds to the structure of the grammar. Analyses for the input validation phase may verify that the code only reads data structures that are designed to hold input data, never reads input data, and writes no data at all. Such analyses may also verify that specific safety checks are applied to specific data structure components. Analyses for the input processing phase may verify that the code never reads additional input and ensures that all data passed to critical functions was appropriately checked in the input validation phase.

The relevant analyses verify a range of properties, from general properties that essentially all applications should exhibit to application-specific properties determined by the specific function that the application implements. Such analyses are, therefore, often profitably specified as combinations of a range of more basic analyses, which is one of the specific goals of our framework.

For the formal presentation of our framework, please refer to our publication [21].

7 Results, Evaluations, and Demonstration

During AMP, we were evaluated on a series of Challenges developed by the evaluation team. These challenges represent real-world binary patching problems related to software flaws and security vulnerabilities.

7.1 Phase I

During the AMP program Phase I, six Challenge problems were distributed. The first two problems are out of scope because they did not include ARM32 binaries. The remaining challenges model flaws on embedded, stripped ARM32 binaries. Each challenge may include multiple patch fragments, e.g., change a function call in a function or change a conditional check on a variable. For all of the ARM32 challenges, CBP operated on the provided stripped binaries. CBP automatically supports (end-to-end) 18 of the 22 patch fragments from the challenges. For the 3 patch fragments we do not support end-to-end, we support the representation and application of the patch fragment internally, but we do not yet support expressing them in the Binary Ninja UI. For the final patch fragment, we do not yet support changing the type of a global variable.

The following table shows the binary change-set results for the ARM32 AMP Challenges 3-6 (Phase I), comparing CBP to the patches produced by the AMP evaluation team. The AMP evaluation team represents the status quo

for binary patching, and we can see from the table that we produce patched binaries with substantially smaller change sets than the evaluation team.

	CBP Patch	**Evaluation Patch**
Challenge 3	1 instruction	12% of ins changed; 50% of ins at different addrs
Challenge 4	2 instructions	85 ins changed (in user code); 14 different basic blocks; library code moved (700+ functions)
Challenge 5	9 instructions	130 ins changed (in user code); 82 different basic blocks; library code moved (4500+ functions)
Challenge 6	62 instructions	156K of original binary changed with recompilation

Table 2. Comparison of CBP Patch and Evaluation Patch

For all the challenges, we delivered assurance analysis artifacts reviews that proved our generated patch correctly implemented the semantics of the patch. For many of the patch fragments, we were able to use CodeHawk's invariants to prove specific relational semantic properties of the original versus patched binary. For example, employing the invariants, we built a reasoned proof of the changes to the state machine represented by the brake light controller of Challenge 3.

The CodeHawk analysis runs in under 12 minutes to analyze the user functions for each of the challenge binaries.

AMP also had a Phase I end evaluation, for which a previous version of CBP, without the Binary Ninja UI plugin, enabled a developer to modify our lifting and automatically create a patch on a stripped binary, with minor live guidance from the CBP team.

7.2 Phase II

At the end of the Phase II of the AMP program (2023Q1), we delivered a full end-to-end CBP platform presented in the context of the Binary Ninja plugin. Our team for the evaluation included STR/BSI, and we collaborated on the workflow for the analyst. The analyst used BSI to find source code matches, and associate source symbols and types to the stripped binary (ARM32). The symbol information is read by CBP (via the BinaryNinja DB), and patching proceeds with the analyst modifying the CBP-lifted C source code. The result was an unqualified success and demonstrated the revolutionary nature of our tools, their maturity, and their potential.

The platform was installed and employed by two independent developers assigned to us by the AMP evaluation team. The evaluation's task was to fix a security vulnerability in the J1939 TP Communication Management system of a heavy vehicle controller. The vulnerability enabled an attacker to overflow memory and deny service.

For this evaluation, we enabled our independent developer team to create a correct patch on the binary in 20 minutes, counting the time from when they started the CBP workflow. The Aarno team did not have to provide any instruction to the independent developers. This was the fastest time to patch recorded for any of the patching teams on AMP during this evaluation. We consider this a revolutionary result, as the developers did not have to investigate the binary at all; the developers stayed at the source code level, and CBP automatically extracted the relevant information from the binary and automatically applied the patch (including finding free space). The CBP UI plugin next presented the relational artifacts to the developers; and they were able to convince themselves that the patch was applied correctly to the binary. Finally, they tested the binary on the target architecture and received the test results expected of a correctly patched binary: maintaining correct behavior and fixing the vulnerability.

The evaluation team stated the following in their debrief:

> "Evaluation went very smooth. All components worked out of the box, documentation was very thorough, and the creation of custom patches/verification was intuitive.
> Overall, the evaluations highlighted several high-level accomplishments
> - Much of the manual lifting effort was able to be supplemented with TA1 (BSI) interoperability

- Patching was done directly in the decompilation view provided
- Fit well into the iterative or cyclic patch development process
- User can formulate different queries to identify the verification results needed
- Verification gives multiple ways of viewing relevant areas”

CBP produced a patch that changed only 10 bytes of the original binary. The evaluation team manually produced a patch that modified 57 bytes of the original binary as a comparison. This demonstrates that CBP can produce minimally-invasive patches automatically, with fewer bytes changed than a patch produced by reverse-engineering SMEs. The patch was also the smallest binary change set of any of the patching performers on AMP. Minimally invasive binary changes enable CBP to produce precise relational analysis and maintain testing and certification efforts on the original binary.

Our Phase II end evaluation effort was selected by AMP’s Program Manager (Sergey Bratus) as an exemplary accomplishment for the program, and was reported up the DARPA chain.

7.3 Phase III

In Phase III, we concentrated on improving our end-to-end, unified UI workflows in Binary Ninja. This included: integration with STR / BSI, binary analysis, patching on lifting, patch application to binary, and assurance analyses. During Phase III, we were awarded a transition contract by STR as a subcontractor on ARPA-H’s DIGIHEALS program. We are working on continuing our AMP-based assured micropatching work in the context of medical devices.

In the final evaluation of AMP (March 2024), we delivered a fully virtualized environment to run our analyses and workflows in Binary Ninja. The delivery included documentation, a patching manual, and walkthroughs of patching examples. The workflow is contained in Binary Ninja, and the operator does not have to run other tools. The operator could change our C lifting of the function to express the patch directly in Binary Ninja. With one press of a button, the semantics of the changes on the lifting were automatically applied to the binary (including a trampoline). The operator ran the relational and patch analysis and reviewed the output. The patch was tested on hardware and passed all tests. The evaluation team noted our success as the only team to 1) provide a unified workflow environment and 2) enable the operator to enact the patch entirely independently.

8 Artifacts

This section presents an overview of the important artifacts produced during the AMP program.

8.1 Open-Source Tools

- CodeHawk Analyzer: https://github.com/static-analysis-engineering/codehawk
- CodeHawk Binary Analyzer: https://github.com/static-analysis-engineering/CodeHawk-Binary
- CodeHawk C Analyzer: https://github.com/static-analysis-engineering/CodeHawk-C
- CodeHawk Patcher [to be released]
- CodeHawk Binary Patcher Binary Ninja Plugin [to be released]

8.2 Publications

- A Unified Algebraic Framework Of Program Analyses [21]
- Assured Micropatching of Race Conditions in Legacy Real-time Systems [7]

8.3 Important Talks and Demonstrations

- CodeHawk Binary Patcher: High Assurance Binary Patching Without a Reverse Engineer, *BlackHat Arsenal 2024* [To appear]. For video, see [13].

9 Lessons Learned

This section details the important lessons learned during the AMP program.

9.1 Binary Analysis and Lifting

Importance of Incorporating Idiomatic Type Usage in Liftings. It was immensely successful for us to partner with STR's BSI TA1 team. The BSI team produces matchings between stripped binary functions and source code, associating source symbols and types within Binary Ninja's intermediate representation. We then incorporate source names and types into our liftings, representing type accesses as the original source would. This enables the patch developer to understand and modify the produced lifting more easily.

Importance of Rigorous Semantics of Analysis Results. A binary analysis produces facts about the binary, and downstream tools must completely and accurately understand the semantics of the results. Thus, we have defined a format and semantics to encapsulate binary analysis results required to enable minimally invasive binary patching. We developed our Patching IR (PIR) to enable multiple binary analysis tools to be used as input to our patcher. We implemented support for the CodeHawk binary analysis frontend and prototype support for extracting analysis information from Binary Ninja and encoding in PIR. If semantics are not rigorously defined, then actions performed on those results, like patching and lifting, could be erroneous.

9.2 Patching

Validated, Best-Effort Patching Enabled Rapid Development, and Expanded Patch Type Support. Other AMP TA2 patching teams attempted to perform changes to a binary in a verified environment. We noticed these teams either abandoned their plan or did not support many patch types. Instead, in our approach the patcher, the tool making changes to the binary, is not a verified component. The patcher makes changes to the binary in a best effort, and then those changes are validated to be correct (or incorrect by the absence of a validation result) by our assurance analyses. This design enabled us to use off-the-shelf compilers (e.g., LLVM) for instruction selection of the new instructions of the patch. Our design enabled rapid development of our tools and support for many patch types.

C Language is Appropriate for Specifying Mircopatches. Expressing a binary micropatch as changes to a C language lifting of the function allowed adequate expressivity in a well-defined semantics. Also allowing us to reuse C analysis tools we have developed in the past. Most security mircopatches can be expressed directly as changes to the C lifting. However, some more complex patches, which stretch the definition of micropatch, are not easily expressed as changes to a C lifting. Continued work should look into supporting multiple representations and techniques for expressing and applying patches. Ideally, it should come with a catalog that informs the user which representation/technique is most appropriate for a given patching need.

Minimally-Invasive Binary Changes. It was essential to produce as small a change set as possible on the binary to enact a required patch. Small change sets enabled precise relational assurance analysis on the binaries (original and patched). Supporting a function's minimal granularity is inadequate to enable precise and understandable assurance analysis results. Our patching tooling reasons about changes on the individual expression and instruction level. Our minimally invasive patches enabled the Galois team to achieve tractable results for their relational change analysis; Galois could not analyze other TA2 performers' patches tractably.

9.3 Assurance

Validate and Abstract Out Implementation Artifacts of Patch Implementation. Our high-level patching workflow does not require the operator to reason about the underlying binary. Our patcher attempts to automatically find free space and create a trampoline if required by the patch (i.e., for insertion patches). However, we do not want the user to have to reason about the trampoline, as it is an implementation artifact added by the patcher that should not affect patch semantics (though it may affect the timing). Thus, our assurance analyses validate and then abstract away the trampoline, representing the patched binary's semantics as if the trampoline is an inlined function call.

Separation of Assurance Concerns. We effectively separated and focused on two important aspects of patch assurance: 1) does a patch's semantics correctly fix the underlying problem, and 2) were the patch's semantics correctly applied to the binary. We demonstrate 1) for important vulnerability types by incorporating CodeHawk C analyzer on the operator's changes to the C lifting. We demonstrate 2) by comparing the modified C lifting to the lifting of the modified binary (validating and abstracting away any trampoline).

9.4 CBP User Interface

Single Pane of Glass. Having a workflow contained completely in a single tool is essential to providing a productive tool. Our Binary Ninja plugin provides an end-to-end workflow for analysis, patching, and assurance. Also, we extract type and source symbol information that may have been added manually in Binary Ninja by the operator. The operator never has to leave Binary Ninja.

Appropriate Representation of Analysis Results. It is difficult to represent assurance analysis results in a manner understandable to a typical developer. We experimented and successfully represented our analysis results as C code and the differences between C code. Instead of making up new representations of differences, we demonstrated differences as diff-style output on C functions. This proved intuitive for the evaluation operations, as they were expected to be high-level language developers. Specifically, comparing user-written code with the lifted code of the patched binary is a powerful and intuitive mechanism for providing assurance that a patch was implemented correctly.

Immediate and Real-Time Feedback Reduces Time to Patch. We successfully reduced the time to run many of our analyses, such that they could be used to provide immediate feedback to the patcher. For example, when modifying the lifted C code to express a patch, we can provide immediate feedback from CodeHawk's C analyzer as to the effects of the patch on the C undefined behaviors such as overflows.

Summary Reports are Helpful for Operators. We produce a summary report detailing the analysis validations and the change to the binary required to implement the patch. The operators from TA3 found our summary report very helpful in quickly understanding what our tool validates and how the binary has changed.

10 Conclusion

In conclusion, the Multifocal Relational Analysis for Assured Micropatching (MRAM) project has achieved significant advancements in the field of binary patching. By developing the CodeHawk Binary Patcher (CBP) platform, we have democratized the process of patching stripped binaries, making it accessible to typical software engineers rather than specialized reverse engineers. Our innovative approach, which includes lifting binaries to an editable form, validating translation steps, and providing assurance artifacts, has not only reduced the costs and errors associated with binary patching but maintains much of the original binary's testing and certifications. The successful implementation of our prototype within the Binary Ninja platform, as well as the positive feedback from independent operators, underscores the efficacy and intuitiveness of our tools.

The MRAM project's accomplishments highlight the potential for rapid, assured binary micropatching without the need for reverse engineering. The transition-ready, independently evaluated workflow demonstrated its capability by achieving the fastest patching times and the smallest changesets in DARPA AMP evaluations. As we move forward, the continued development and integration of the CBP platform in real-world applications, such as medical device security under the ARPA-H DIGIHEALS program, promises to further enhance the security and reliability of critical systems. The MRAM project's contributions lay a solid foundation for future research and development in binary patching, ensuring that organizations can address vulnerabilities in legacy systems with greater confidence and efficiency.

References

[1] Aarno Labs. [n. d.]. CodeHawk Tool Suite. https://github.com/static-analysis-engineering. https://github.com/static-analysis-engineering

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA.

[3] Gogul Balakrishnan and Thomas W. Reps. 2004. Analyzing Memory Accesses in x86 Executables. In *Compiler Construction, 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2985)*, Evelyn Duesterwald (Ed.). Springer, 5–23. https://doi.org/10.1007/978-3-540-24723-4_2

[4] Gogul Balakrishnan and Thomas W. Reps. 2007. DIVINE: DIscovering Variables IN Executables. In *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4349)*, Byron Cook and Andreas Podelski (Eds.). Springer, 1–28. https://doi.org/10.1007/978-3-540-69738-1_1

[5] Sergey Bratus, J. Adam Crain, Sven M. Hallberg, Daniel P. Hirsch, Meredith L. Patterson, Maxwell Koo, and Sean W. Smith. 2016. Implementing a vertically hardened DNP3 control stack for power applications. In *Proceedings of the 2nd Annual Industrial Control System Security Workshop, ICSS 2016, Los Angeles, CA, USA, December 6, 2016*. ACM, 45–53.

[6] Sergey Bratus, Lars Hermerschmidt, Sven M. Hallberg, Michael E. Locasto, Falcon Momot, Meredith L. Patterson, and Anna Shubina. 2017. Curing the Vulnerable Parser: Design Patterns for Secure Input Handling. *login Usenix Mag.* 42, 1 (2017).

[7] Rik Chatterjee, Ben Karel, Ricardo Baratto, Michael Gordon, and Jeremy Daily. 2024. Assured Micropatching of Race Conditions in Legacy Real-time Embedded Systems. In *First Workshop on Real-Time Autonomous Systems Security (RTAutoSec)*.

[8] Ravi Chugh, Jan W Voung, Ranjit Jhala, and Sorin Lerner. 2008. Dataflow analysis for concurrent programs using datarace detection. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 316–326.

[9] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

[10] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski (Eds.). ACM Press, 84–96. https://doi.org/10.1145/512760.512770

[11] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 313–324. https://doi.org/10.1145/2642937.2642982

[12] Paul Black Henny Sipma. 2020. Scalable Sound Static Analysis of Real-world C Applications using Abstract Interpretation. https://cps-vo.org/node/70218 High Confidence Software and Systems Conference, Sept 2020.

[13] Aarno Labs. 2024. *CodeHawk Binary Patcher: BlackHat Arsenal 2024*. Youtube. https://www.youtube.com/watch?v=QgrdOLPyaLQ

[14] Falcon Momot, Sergey Bratus, Sven M. Hallberg, and Meredith L. Patterson. 2016. The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them. In *IEEE Cybersecurity Development, SecDev 2016, Boston, MA, USA, November 3-4, 2016*. IEEE Computer Society, 45–52.

[15] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. 2005. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. In *Proceedings of the 2005 International Workshop on Mining Software Repositories* (St. Louis, Missouri) *(MSR '05)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/1083142.1083143

[16] George C. Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, Monica S. Lam (Ed.). ACM, 83–94. https://doi.org/10.1145/349299.349314

[17] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2304)*, R. Nigel Horspool (Ed.). Springer, 213–228. https://doi.org/10.1007/3-540-45937-5_16

[18] Terence Parr and Kathleen Fisher. 2011. LL(*): the foundation of the ANTLR parser generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 425–436.

[19] pir [n. d.]. Patching Intermediate Representation. https://github.com/static-analysis-engineering/CodeHawk-Binary/tree/master/chb/ast/doc.

[20] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1384)*, Bernhard Steffen (Ed.). Springer, 151–166. https://doi.org/10.1007/BFb0054170

[21] Martin Rinard, Henny Sipma, and Thomas Bourgeat. 2021. Work in Progress: A Unified Algebraic Framework Of Program Analyses. In *LangSec Workshop at IEEE Security and Privacy* (Virtual).

[22] Martin C. Rinard and Darko Marinov. 1999. Credible Compilation with Pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verification*.

[23] Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*. IEEE Computer Society, 186–199. https://doi.org/10.1109/CSF.2010.20

[24] Jiasi Shen and Martin Rinard. 2017. Robust programs with filtered iterators. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*, Benoît Combemale, Marjan Mernik, and Bernhard Rumpe (Eds.). ACM, 244–255.

[25] Karen Zee, Viktor Kuncak, and Martin C. Rinard. 2008. Full functional verification of linked data structures. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008.* ACM, 349–361.

[26] Kaizhong Zhang, Rick Statman, and Dennis Shasha. 1992. On the editing distance between unordered labeled trees. *Inform. Process. Lett.* 42, 3 (1992), 133–139. https://doi.org/10.1016/0020-0190(92)90136-J