Match & Mend: Minimally Invasive Local Reassembly for Patching N-day Vulnerabilities in ARM Binaries

Sebastian Jänich
LMU Munich
Munich, Germany
sebastian.jaenich@lmu.de

Merlin Sievers
LMU Munich
Munich, Germany
merlin.sievers@lmu.de

Johannes Kinder
LMU Munich
Munich, Germany
johannes.kinder@lmu.de

***Abstract*—Low-cost Internet of Things (IoT) devices are increasingly popular but often insecure due to poor update regimes. As a result, many devices run outdated and known-vulnerable versions of open-source software. We address this problem by proposing to patch IoT firmware at the binary level, without requiring vendor support. In particular, we introduce minimally invasive local reassembly, a new technique for automatically patching known (n-day) vulnerabilities in IoT firmware. Our approach is designed to minimize side effects and reduce the risk of introducing breaking changes. We systematically evaluate our approach both on 14 vulnerabilities within the controlled environment of the MAGMA benchmarks, as well as on 30 real-world Linux-based IoT firmware images from the KARONTE dataset. Our prototype successfully patches 76% of targeted vulnerabilities in MAGMA and 96% in the firmware dataset.**

***Index Terms*—Firmware, Patching, Binary Rewriting**

1. Introduction

Internet of Things (IoT) devices, from WiFi routers over smart cameras to robot vacuums, are widely deployed in homes and businesses. Their firmware usually combines open-source components with some proprietary code implementing device- or application-specific functionality. A popular and flexible architecture design choice is Linux for 32-bit ARMv7, which can form a basis for minimal to complex embedded systems distributions [1].

Especially for low-cost devices, security is often not a priority, so they are insufficiently hardened and receive updates only irregularly or never at all. This leads to IoT devices running outdated software containing known vulnerabilities and potentially becoming targets or entry points for attackers [2]. This issue is wide-spread; by scanning 16 million home networks, Kumar et al. [3] found 40% of households worldwide to have at least one vulnerable smart device. While recent policy initiatives (e.g., in the European Union [4]) require vendors to provide updates, direct sales of low-cost devices to end users circumvent even existing safety standards, so it is highly likely that such devices will continue threatening the security of the networks they are deployed on.

In this paper, we focus on a purely technical solution to vulnerable IoT devices, based on the idea of simply patching any known (n-day) vulnerabilities *without support from the manufacturer*. Key to our approach is a highly localized static binary reassembly algorithm that patches just the vulnerability, while ensuring consistency and leaving the rest of the firmware untouched. Our guiding principle in designing our solution is to avoid introducing any breaking changes to the firmware.

Clearly, an initial obstacle is to extract the firmware from the device. Possible access paths depend on the specific device, but they follow common patterns [5]. The fully patched firmware then has to be reinstalled on the device in a similar fashion. While some devices do use signature schemes that prevent installing custom firmware (e.g., Amazon Echo, Apple Homepod), many update mechanisms are not secured [6], [7]. Additionally, physical access opens many more possibilities, e.g., through exposed debug interfaces such as UART and JTAG [8], [9], or via backup mechanisms like flashing from an SD card [10]. In particular, we argue that on low-cost devices the same type of lax security posture that makes our approach to patching *necessary* also makes it *feasible*.

The ideal solution would be to obtain the original source code and build environment for the device, update any included open-source components to fix known vulnerabilities, rebuild the firmware and upload it to the device. However, without the cooperation of the manufacturer, we lack critical components for building the firmware. While the code of the open-source components is available, we will miss any custom closed-source applications added by the manufacturer; any custom patches that were applied to a standard release of the open-source components; and the specific build configuration including compile-time switches. For instance, OpenSSL has over 20 compile-time options that enable or disable various features and protocols. If we were to replace the entire OpenSSL package because of a vulnerability, we would have to reconstruct the exact same combination of options to reduce any differences in behavior, including resource consumption; even for individual functions, this is a challenging problem [11].

Therefore, we see *micro-patching* of known vulnerabilities at the binary level as a promising path to improve the

Published in 11th IEEE European Symposium on Security and Privacy, Lisbon, Portugal, July 6 – 10, 2026

security of low-cost devices. In this paper, we focus on the problem of flexibly rewriting and reassembling the patch in a target binary; a complete end-to-end solution additionally requires applying existing techniques for identifying open-source libraries in firmware [12] and finding known vulnerabilities [13], [14]. In our prototype, we employ Intel's publicly available CVE-bin tool [15] for this purpose.

Binary rewriting is a challenging problem in itself. Existing approaches involve either static [16]–[19] or dynamic [11], [20]–[22] techniques to modify the binary code. A survey by Wenzl et al. [23] finds that work so far predominantly focuses on the x86 architecture, leaving 32-bit ARM binaries, typical for IoT devices, with fewer solutions. To the best of our knowledge, in this paper we are the first to propose a minimally invasive local reassembly approach for an automated patching process. Crucially, we require a persistent solution where the code is written into the binary itself. Unlike dynamic approaches such as OSSPatcher, which pioneered runtime patching on Android [11], we cannot rely on intercepting the loading or linking process on general IoT devices, where the global flow of control may be unknown. Our problem is also harder than other common static binary rewriting tasks: the code being rewritten can interact heavily with the rest of the binary, including references to functions and global variables.

However, our setting offers the unique advantage that we do have *some* source code for the binary we are rewriting; we just may not have the exact same source code, and the compilation of the target is outside our control. We leverage our knowledge about the code to identify functions to be patched and match basic blocks such that we can reconstruct code and data references.

Match&Mend combines static binary analysis and local reassembly of ARM32 binary code to produce automated, minimally invasive micropatches. This yields a practical local reassembly pipeline, transplanting instructions from one binary into another with recomputed relative offsets and relocations avoiding global relinking or full binary recompilation, thereby reducing layout perturbation and side effect risk. In summary, the main contributions are:

- A novel concept for local reassembly by resolving a scoped symbolic mapping of instruction-level references (e.g., PC-relative loads, literal pools, GOT/PLT entries, and function branch targets) to concrete addresses/offsets in the host binary (§4.2).
- An algorithm for minimally invasive local reassembly of micro-patches in ARM binaries, to effectively patch known vulnerabilities in IoT firmware (§4.3).
- The implementation of this algorithm in *Match & Mend*, a working prototype system for automated binary patching in ELF binaries (§5).

We thoroughly evaluate our prototype on both the MAGMA benchmark and a dataset of real-world firmware images, demonstrating that Match & Mend can successfully patch firmware vulnerabilities in practice (§6). All source code and data are available on GitHub.[1]

1. https://github.com/lmu-plai/match-and-mend

2. Problem Definition

We begin by outlining the scope of the automated binary patching problem (§2.1) and identifying the core challenges (§2.2).

2.1. Scope

Binary patching is the process of fixing a vulnerability in a binary executable without changing its intended functionality. In the following, we informally define our interpretations of relevant terms to help explain our approach. We consider a *vulnerability* in a binary to be the set of instructions and data that corresponds to the minimal set of statements in source code that enables the attacker to potentially exploit the binary, e.g., an unchecked access of an array in C. For the same source code, there can be many instances of the vulnerability in binary form. Similarly, a *patch* in binary form is the set of instructions and data that correspond to the minimal set of statements in source code that need to be modified or added to eliminate the vulnerability without changing the originally intended behavior, e.g., adding a check before accessing an array in C.

The scope of our binary patching problem is defined as follows: there is a Linux-based IoT firmware that contains open-source libraries. Among these libraries is one with a known and documented n-day vulnerability. Furthermore, a patched version of the library's source code is available. The task is to determine how to patch the vulnerable binary in an automated and minimally invasive way.

2.2. Challenges

The main goal of existing rewriting techniques is to enable instrumentation of binaries for various purposes such as fuzzing, hardening, or analysis. The core challenges that need to be solved to correctly rewrite a binary, such as pointer detection, telling code from data and handling indirect control flow transfers, all without introducing breaking changes or too much overhead, can be solved using existing techniques. We identified new challenges specific to the problem of binary patching:

(C1) Patch Identification: To be able to correctly patch a vulnerability, a system needs to identify a patch in binary code that removes the vulnerability.

(C2) Vulnerability Identification: The binary code containing the vulnerability has to be identified. Even when the vulnerability is known in source code form, the exact source code and flags for compilation generally are not.

(C3) Control Flow Integration: The control flow of the patch has to be integrated into the control flow of the vulnerable code. This involves jump and call redirection to connect the patch with the target binary.

(C4) Data Flow Integration: Also the data flow of the patch needs to be integrated into the data flow of the vulnerable code. This may include adding new data to the binary.

(C5) Encoding Modified Instructions: During reassembly, instructions and their sizes (depending on encoding format) might change. However, due to relative addressing, the relative distances between instructions need to be preserved to not introduce breaking changes.

3. Overview

Now we introduce an illustrative example to give an overview of our approach (§3.1). It is structured into three main phases, *Analysis* (§3.2), *Matching* (§3.3), and *Local Reassembly* (§3.4). The first two phases serve as a prerequisite for the third, which is the centerpiece of the system. Challenges **C1** and **C2** are addressed in the initial phases by identifying an over-approximation of the patch and the vulnerable code. Challenges **C3**, **C4**, and **C5** are handled in the third phase by the local reassembly algorithm utilizing the matching results and control- and data flow analyses from the earlier phases.

3.1. Running Example

Figure 1 shows the different components of our system. For example, consider a smart camera running a Linux-based firmware that no longer receives (security) updates from the manufacturer. For processing PNG files, the application code in the firmware relies on the open-source library `libpng 1.0.65`, an outdated and known-vulnerable version of the library. Consider a fictional vulnerability (loosely inspired by CVE-2015-8540 in `libpng`) caused by an improper bounds check of an index variable. Specifically, the variable may not be zero or smaller to prevent an out-of-bounds write. However, the vulnerable version only checks that the value does not equal zero, possibly due to misreading the signedness of the variable. Assume that this vulnerability is patched in `libpng 1.0.66` by fixing the check and additionally assigning a fallback value to it, which was also missing in the vulnerable version. In the following, we will explain how our system identifies this patch in binary form and reassembles it in the vulnerable binary.

The input for our patching system consists of two binaries and the name of the affected function. The first binary is the vulnerable library obtained from the firmware binary, `libpng 1.0.65`, which was compiled by the original manufacturer with unknown settings. The second binary is the patched version of the library, `libpng 1.0.66`, which we compile ourselves from its publicly released source code. We use this self-compiled `libpng 1.0.66` to solve the challenge of identifying the parts of the binary that need to be rewritten to patch the vulnerability.

3.2. Analysis Phase

Our approach begins with the analysis phase ①, which lays the foundation for the following steps. The system recovers the interprocedural control and data flow graphs (CFG and DFG, see §4.1) for both binaries. We then use the name(s) of the affected function(s) to identify the corresponding code in the vulnerable binary, by first looking for a matching symbol. If no symbol can be found, we identify the functions using binary code similarity analysis between the patched and vulnerable binaries. Figure 1 shows a simplified version of the subgraph of the CFG for the affected function in the vulnerable binary of `libpng 1.0.65` and the corresponding subgraph of the affected function in the patched binary of `libpng 1.0.66`.

3.3. Matching Phase

In the matching phase ②, we leverage the control- and data flow information gathered in the analysis phase to identify the instructions and data in the patch that need to be reassembled in the vulnerable binary. A binary code similarity analysis determines the basic blocks of the vulnerable function that match the basic blocks of the patched function. The set of basic blocks that are not part of a *perfect match* (defined in §4.2) in the patch binary is an over-approximation of the actual patch. Determining this set, which will be written to the vulnerable binary, solves challenge **C1**. The set of basic blocks not part of a perfect match in the vulnerable binary are then dually an over-approximation of the vulnerability, which will be replaced by the patch, solving challenge **C2**.

In Figure 1, these basic blocks are visualized by the green nodes in the CFG of the patch binary of `libpng 1.0.66`. The gray nodes in both CFGs are the matched nodes and the red nodes are unmatched nodes in the vulnerable binary of `libpng 1.0.65`. The matching reveals that both the vulnerability and the patch consist of two basic blocks each (note that the number of blocks may also differ). If there are no perfectly matched blocks, then all basic blocks of the function will be reassembled. This approach minimizes the amount of code being reassembled while introducing no additional effort compared to reassembling the entire function.

Additionally, the system needs to consider the dependencies between nodes to correctly integrate the patch. Control flow dependencies (**C3**) are represented by the CFG; for data flow dependencies (**C4**), we define the concept of references and matches of references (see §4.2). The last step of this phase is to systematically match such references based on the perfect matches of basic blocks such that there is a one-to-one correspondence between references in the vulnerable and patched versions (see Algorithm 1).

3.4. Local Reassembly Phase

In the final reassembly phase ③, the system integrates the identified instructions and data from the patch binary into the vulnerable binary (see §4.3). The requirements of this phase are inherently tied to the underlying processor architecture, in our case 32-bit ARM, common with IoT devices. This architecture features both the ARM instruction set (consisting of 32-bit instructions) and the Thumb instruction set (consisting of mostly 16-bit instructions). Runtime

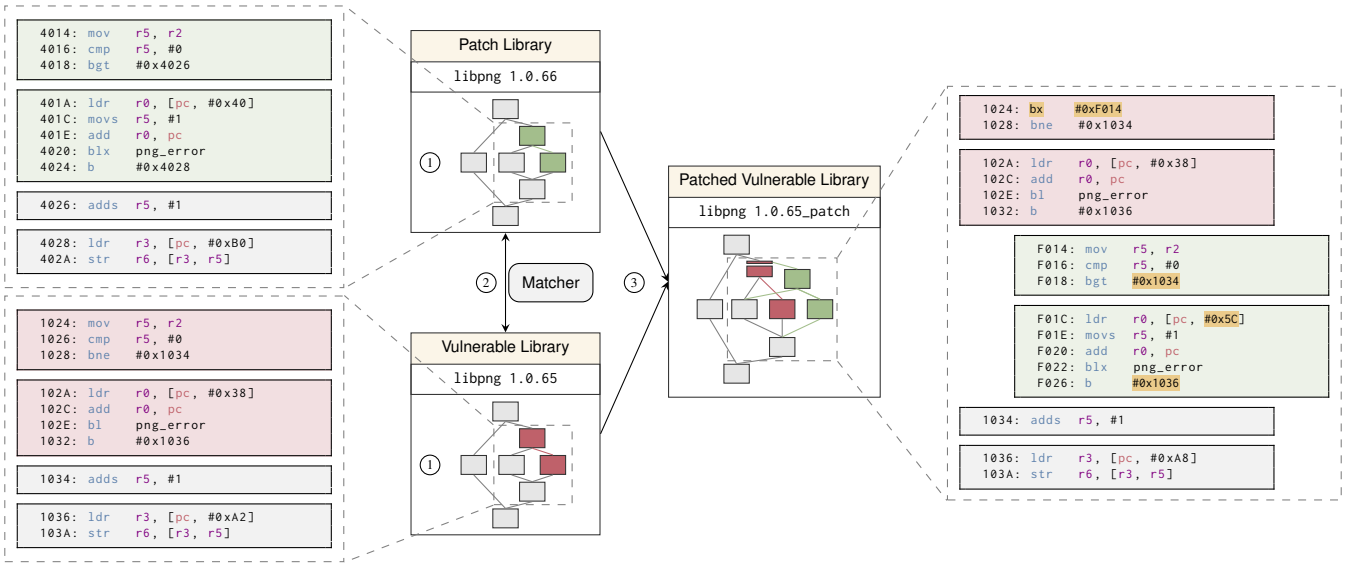

Figure 1. Schematic overview of the patching process, using the fictional example of a vulnerable `libpng 1.0.65`, designed to concisely showcase the challenges and solutions of minimally invasive local reassembly. The vulnerability is patched in `libpng 1.0.66`. ① We identify vulnerable and patched functions in the binary and construct control and data flow graphs; ② we match basic blocks between the functions using binary diffing and use this information to match references to code and data; ③ we transplant the code from the patched to the vulnerable version, adjusting all code and data references as needed and fixing up shifts from differences in addressing modes.

behavior may include switching between those instruction sets. Furthermore, PC-relative addressing is essential in both instruction sets, as instructions generally have only limited space for encoding immediate address values. The first requirement for successful reassembly is the correct disassembly of machine code as either Thumb or ARM instructions. This is challenging, as the mode of decoding can depend on values computed at runtime [24]. While some prior work addresses this problem, perfect disassembly cannot generally be guaranteed [25].

The second requirement is the ability to handle the different instruction sizes possible in Thumb. When not all operands can be encoded in 16 bits during reassembly, the size of a modified instruction may change. For instance, in Figure 1 we can see a conditional jump (`bgt`) to `0x4026` in the first green basic block of the patch library at address `0x4018`. The relative distance of this jump can be encoded in a 16-bit instruction. However, when this instruction is rewritten to location `0xF018` in the vulnerable binary about to be patched, the distance increases and the instruction has to be rewritten to a length of 32 bits. As a result, all following instructions have to be shifted by two bytes, requiring further adjustments to relative addressing. For example, consider the load instruction at address `0x401A` in the patch library, which reads data from a PC-relative location. Due to shifts in relative addressing, the offset from the PC value has to be adjusted accordingly, as shown at address `0xF01C` in the rewritten library. Match & Mend transparently systematically tracks all required shifts and adjusts offsets as necessary, solving challenge **C5**.

After this phase is finished, the system outputs the rewritten vulnerable binary. It now includes the patch and all necessary changes to fix the vulnerability and can be tested for validation.

4. Automatic Binary Patching

In this section, we introduce the core concept and algorithm for local reassembly, solving challenges **C3** to **C5**. After defining the usual terminology for program structures (§4.1), we introduce our notions of references and matches (§4.2), before describing the algorithm itself and how it addresses the challenges (§4.3).

4.1. Preliminaries

We introduce notation to enable us to define the concept of references and perfect matches of basic blocks.

Definition 1 (Basic Block) *A basic block of a program P is a sequence of consecutive instructions in which flow of control enters at the beginning and leaves at the end, without halt, possibility of branching (conditional jumps), or return, except at the end. A call (`bl`/`blx`) to a function that is expected to return does not end the basic block. We denote the set of all basic blocks of P by P_{BB}.*

Given a program P, we denote the set of all *instructions* of P by P_{Ins}. A *memory location* of P can be a register, a stack variable, heap variable or a global variable. We denote the set of all memory locations of P by P_{Mem}. If I is an instruction of a program P, then we denote with B_I the basic block containing I.

Definition 2 (Definition and use of memory locations) *Let v be a memory location of P. We say that v is* defined *in instruction $I \in P_{Ins}$ if and only if I writes to v. We say that v is* used *in instruction $I \in P_{Ins}$ if and only if v is read by I.*

Definition 3 (Control-Flow Graph) *A control flow graph $G_{CFG} = (P_{BB}, E_{CFG})$ of a program P is a directed graph with a set of nodes P_{BB} and a set of edges $E = P_{BB} \times P_{BB}$. There is a directed edge (B, C) from node B to node C, if and only if C can immediately follow B in some execution sequence of P:*

(i) Jump edge: there is a conditional or unconditional jump from the last statement of B to the first statement of C.
(ii) Call edge: there is a call from an instruction in B to the first instruction of C.
(iii) Fall-through edge: C immediately follows B in the program P and B does not end in an unconditional jump or return.

We say that node B is a predecessor *of C and node C is a* successor *of B.*

Definition 4 (Data Flow Graph (DFG)) *A data flow graph $G_{DFG} = (P_{Ins}, E_{DFG})$ of a program P is a directed acyclic graph with a set of nodes P_{Ins} and a set of edges*

$$E = \{(I, J)_v \mid I, J \in P_{Ins} \text{ and } v \in P_{Mem}\}.$$

There is directed edge $(I, J)_v$, i.e., a data dependence from I to J with respect to a memory location v if and only if there is a non-null path p in the CFG of P from B_I to B_J, with no intervening definition of v and I contains a definition of v and J a use of v.

The local reassembly algorithm in §4.3 is designed to be accurate under the assumption of precise CFGs and DFGs. While recovering an exact CFG or DFG of a binary program is a hard and undecidable problem [26]–[28], disassembly of compiler-generated code has become increasingly reliable [29]. We give a detailed explanation of the implementation in §5 and an empirical evaluation of our prototype in §6.

4.2. Locally Scoped Symbolic Mapping

Local reassembly of a binary patch implies adding its basic blocks to the CFG of the host binary. To correctly add and integrate these nodes, we must identify all incoming and outgoing edges and their corresponding nodes in the DFG. This process relies on generating a locally scoped symbolic mapping that resolves elements such as PC-relative loads, literal pools, and GOT/PLT entries into concrete addresses or offsets specific to the host binary. To achieve this, we define references and perfect matches of basic blocks.

Definition 5 (Reference) *Let P be a program and $r = (I, J)$, with $I, J \in P_{Ins}$, a pair of instructions of the program. We say that r is a reference if and only if one of the following conditions holds:*

- *There exists an edge $e = (I, J)_v$, with $I, J \in P_{Ins}$ and $v \in P_{Mem}$, in the data flow graph of P, then we call r a data reference.*
- *I is a call or a conditional or unconditional jump instruction and there exists a non-fallthrough edge $e = (B_I, B_J)$, with $B_I, B_J \in P_{BB}$, of the control flow graph of P, then we call r a control reference.*

We call I the source and J the destination of r.

For minimally invasive patching, we wish to rewrite as few instructions as possible; and consequently, as few basic blocks as possible. To solve this problem, we need to decide whether two basic blocks are semantically equivalent and do not need to be rewritten. A fully formal definition of equivalence for basic blocks with respect to the formal semantics of machine code is out of scope for this paper and ultimately unnecessary for our purpose. Instead, we define a pragramatic notion of perfect matches between basic blocks, based on the intuition that two basic blocks should be equivalent if they have the same instructions, allowing for variation in compiler-assigned addresses.

Definition 6 (Perfect Match of Basic Blocks) *Let P_1, P_2 be two programs and B_{P_1}, B_{P_2} two basic blocks of P_1 and P_2, respectively. We say that B_{P_1} and B_{P_2} are a perfect match if and only if the sequence of instructions in B_{P_1} matches that in B_{P_2}; i.e., the sequences have the same length and the i-th instructions have the same opcodes and operands, except for operands used as addresses.*

Definition 7 (Match of References) *Let $r = (I, J)$ be a reference in program P_1 and $r' = (I', J')$ a reference in program P_2. We say that r and r' are a match if and only if at least one of the following conditions holds:*

- *B_I and $B_{I'}$ are a perfect match of basic blocks.*
- *B_J and $B_{J'}$ are a perfect match of basic blocks.*

If $r = (I, J)_v$ is a data reference, then additionally $r' = (I', J')_w$ needs to be a data reference and v and w have to be the same memory location.

These definitions allow us to define our rewriting algorithm (in pseudocode) in the following.

4.3. Local Reassembly Algorithm

The local reassembly algorithm (Algorithm 1) takes the identified basic blocks and references and reassembles them in the vulnerable binary. The algorithm runs for each affected function and in each function processes the instructions one by one. The shown algorithm therefore only shows the processing for one function, but the algorithm is simply repeated for each affected function in the patch.

Before beginning, we designate a target location for the patch in the vulnerable binary. For this we create space by extending an executable load segment or, if necessary, by adding a new section `.patch` in the vulnerable binary. This section can be extended as needed to accommodate future patches whenever a new one is added. To redirect the control

```
input: ins (Instructions of function)
output: Rewritten function
insNew ← AdjustReference(ins);
if ref.dest not in function then
    refVuln ← GetMatchedRef(ref);
    if ref is control reference then
        if refVuln is None then
            newDest ← GetFreeLocation();
            insNew ← Replace(ins, newDest);
            Save(newDest, ref.dest);
        else
            insNew ← Replace(ins, refVuln);
        Reassemble(insNew);
    if ref is data reference then
        if refVuln is None then
            data ← LoadDataFrom (ref.dest);
            WriteData(newDest, data);
            bs ← BackwardSlice(ins, newDest);
            tuples ← Solve(bs.constraints);
        else
            bs ← BackwardSlice(ins, refVuln);
            tuples ← Solve(bs.constraints);
        for (memLoc, value) ∈ tuples do
            WriteData(memLoc, value);
        Reassemble(insNew);
else
    Reassemble(insNew);
```

Algorithm 1: Local reassembly algorithm in pseudocode.

flow to this new space, we insert a branch instruction at the point where the patch needs to start. In the following, reassembling in the vulnerable binary means reassembling at this newly created region in the extended segment. At this point, we can start the local reassembly algorithm.

Shifts. The ARM32 Thumb instruction set consists of instructions of 16 and 32 bits length. As described in challenge **C5**, encoding a longer distance in a jump might lead to an increase of the instruction size. Similarly, this also applies to instructions like the `ldr/str` instructions. This results in a shift in the relative distance between surrounding instructions. Instructions and data are often intermixed due to optimizations for space and execution efficiency and the algorithm needs to ensure that data is not overwritten by the shift in instructions. To address this, the algorithm performs two steps, both defined within the function `AdjustReference()`. Step one is to track the potential candidates that would be affected by a shift. The second step is to adjust references to data inside the patch that might be overwritten, by increasing the relative distance of the reference and effectively moving the data further away such that there is room for shifts.

For example, instruction `ldr r1, [pc,#0x18]` in Figure 2 reads data from a memory location that is located inside the patch directly next to instructions. If one of the instructions between the load instruction and the memory

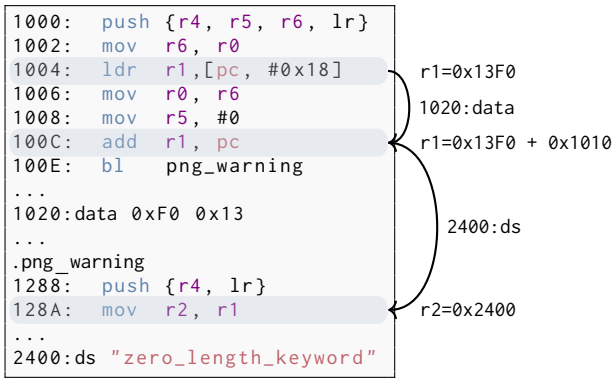

Figure 2. Backward data flow slice computed from a memory access instruction. This slice captures the relevant definitions and uses needed to reconstruct equivalent data flow when reassembling the basic block in a different binary.

location is reassembled from a 16-bit to a 32-bit instruction, the relative distance of the load instruction will be increased, and the data would be overwritten. To prevent this, the algorithm shifts the memory location with its data to a location further away by increasing the value `0x18` and tracks this new location.

Reference Handling. The algorithm resolves the symbolic mapping of all references of the patch such that the patch is integrated seamlessly into the vulnerable binary. We distinguish whether the destination of the reference lies inside or outside the function. For references with their destination *inside* the function, no further modifications are necessary, as all addressing is relative. For references with their destination *outside* the function, the algorithm again distinguishes between *control* references and *data* references.

In case of a control reference (**C3**) the algorithm first invokes the function `GetMatchedRef()`, which returns the matched reference, if there is one (e.g., another function in the binary). Changing control references to their matched destination connects the patch with the vulnerable binary. The function `Replace()` updates the reference's destination to the matched destination address. Then the algorithm reassembles the updated instruction in the vulnerable binary. If there is no matched reference, this means that the destination of the reference is not part of the vulnerable binary. By construction, it is thus part of the patch and will be added to the processing queue, allowing the algorithm to reassemble this function as well. Until then, the algorithm modifies the destination of the reference to point to a free location in the vulnerable binary, saving the new address to be used later as a starting point for the affected function.

For handling of data references (**C4**), consider the assembly code in Figure 2. Assume that the instructions and data from address `1000` to address `1040` form the patch that needs to be reassembled in the vulnerable binary. There is a data reference $r = (I, J)_v$ where I is the instruction at address `100C` and J is the instruction at address `128A`.

```
1004:  ldr  r1,[pc, #0x18]
                        r1  = COPY [0x1020:data]

100C:  add  r1, pc
                        $t1 = INT_ADD 0x100C, 4
                        r1  = INT_ADD r1, $t1

128A:  mov  r2, r1
                        r2  = COPY r1
```

Figure 3. Instructions and their intermediate representation in the backward slice computed in Figure 2.

The memory location v is the global variable `2400:ds`. Depending on whether the reference, i.e., the global variable, is matched or not, there are two different paths the algorithm can take.

Matched Data References. If the reference is matched, the algorithm needs to reassemble the instruction such that the matched memory location (in our example the matched global variable) of the vulnerable binary is used. To do this, function `BackwardSlice()` calculates a backward slice starting from the destination of the reference. It traces all instructions that read from the memory location, going backward until it reaches the instruction that writes to it (its definition). The result includes all instructions that use or define the memory location of the reference, as well as all instructions that use or define related memory locations encountered during the slice. Figure 3 shows the backward slice for the example data reference. In this case it only contains three instructions. The affected memory locations are the registers `r1`, `r2` and `1020:data`

In a next step, the list of instructions together with the affected memory locations serves as input for the function `Solve()`. The function translates the given instructions into an intermediate representation (IR) so that all side effects of the operations are captured. During the translation all used addresses are replaced with the corresponding addresses intended for the reassembled instructions. The function then adds the constraint that the matched memory location is the new destination of the reference.

Figure 3 shows the translation of the instructions in the backward slice into IR. Let us assume that the location in the newly created region where the instruction `add r1,pc` will be rewritten to is `0x130C` and that the matched global variable of `0x2400:ds` is located at `0x2500`. Then the following equation system is constructed that encodes the relationships among the relevant memory locations:

$$
\begin{aligned}
r1_a &= \texttt{0x1020:data} \\
t1 &= \texttt{0x130C} + 4 \\
r1 &= r1_a + t1 \\
r2 &= r1 \\
r2 &= \texttt{0x2500}
\end{aligned}
$$

Solving this yields `r2=0x2500` and `0x1020:data=0x14F0`, which means that the value of the memory location `0x1020:data` has to be changed to `0x14F0`. Note that in Thumb mode the PC register always points to the address four bytes after the current instruction.

The memory locations and the corresponding new values that need to be changed are returned as a list of `tuples`. The algorithm reassembles the instruction and writes the new values to the corresponding memory locations with the function `WriteData()`. In our example the only memory location that needs a new value is `0x1020:data`.

Unmatched Data References. In case the data reference is not matched, we assume that the value of the memory location of the reference is not part of the vulnerable binary. This may be due to the patch adding a previously unused string to the function, for instance. Therefore, the algorithm calls `LoadDataFrom()` to load data from the destination of the reference. In the next step, this data is written to a new destination, `newDest`, in the extended or added segment. The new destination is positioned at a distance from the instruction to provide sufficient space for rewriting the remaining instructions of the patch. Following our example in Figure 2, this means `0x2400:ds` has no match and the string `"zero_length_keyword"` is loaded from memory location `0x2400:ds` and written to a new memory location in the extended segment of the vulnerable binary. With this newly created memory location, the algorithm can now proceed as before, calculating a backward slice and using the constraint solver to adapt all memory locations such that address of the newly added data is reached.

If there is no reference or a reference with destination inside the function, the instruction will be reassembled in the vulnerable binary. An example of this can be seen in Figure 2. The instruction at address `0x1002` has no reference and will therefore be reassembled to the vulnerable binary.

The algorithm is designed to work correctly under the assumption that the control flow graph and data flow graph are complete and correct. In the following section we will present the implementation of the system and show how we deal with the fact that in general it is not possible to obtain complete and sound control flow and data flow graphs.

5. Implementation

We now introduce our implementation in the prototype tool Match & Mend. After reviewing the frameworks we employ in our prototype (§5.1), we present the implementation following the same phases as in the design, i.e., analysis (§5.2), matching (§5.3), and local reassembly (§5.4).

5.1. Tools and Frameworks

For our prototype, we require tools that are able to work with ELF binaries on the ARM32 architecture. A number of static binary rewriting tools from the literature would in principle be eligible to implement our approach, and we will discuss these in §8. In this work, we chose to use the angr [30] binary analysis framework to disassemble binaries and to implement our custom analysis and transformations,

and patcherex [31] to modify the binary and reassemble instructions. Finally, we use lief [32] to modify the structure of ELF files and create sufficient space to add patches to vulnerable binaries.

5.2. Analysis

The prototype loads both the vulnerable library binary and the self-compiled patch binary in angr and recovers CFGs for both. Given the size of the binaries and the complexity of constructing an interprocedural CFG for an entire program, we execute the static and fast, but sometimes imprecise, `CFGFast` analysis provided by angr for this purpose. Because our local reassembly algorithm relies on accurate CFGs for the affected functions, we choose to improve precision locally for those by executing `CFGEmulated`. This computationally more expensive but also more precise analysis recovers the intraprocedural CFG using symbolic execution. In the affected functions, we follow up CFG creation by creating the data flow graphs (DFGs) using the `DDG` analysis of angr.

After the analysis phase, the system has produced an interprocedural CFG for both binaries, as well as DFGs and intraprocedural CFGs for each affected function. It also maintains a list of references for each affected function, which is used during the matching phase. Note that, should additional functions be identified as affected and requiring patching during analysis, the analysis phase will be repeated for these, generating precise control- and data flow graphs.

5.3. Matching

To identify the instructions and data that need to be reassembled, we use the BinDiff [33] implementation of angr to match functions and basic blocks between the vulnerable and the patched version of the open-source library (*Matcher* in Figure 1). We are particularly interested in the basic blocks in the affected functions. The resulting matches of basic blocks are then filtered so that only perfect matches of basic blocks (see §4.2) remain. While BinDiff performs well in most cases, matching can be challenging when binaries have been built with significantly different optimization settings, e.g., when functions have been inlined or loops unrolled [34]. Recent work in the area of binary code similarity analysis promises to improve such shortcomings using machine learning techniques [35]–[41]. We will investigate the integration of improved matching mechanisms in future work.

At this point, the system has compiled a list of perfect matches and identified basic blocks for the affected function in each binary (**C1** and **C2**). They are ordered in ascending address order and the first basic block that is not a perfect match is the starting point for the local reassembly algorithm. Consequently, the last block that is not a perfect match is the end point of the patch. Everything in between forms the patch that needs to be integrated into the vulnerable binary (even if it is a perfectly matched block).

Besides the identification of the patch, the matching of the basic blocks also serves to match references. The prototype compares the references of the vulnerable and the patched version of the open-source library and checks if either their origin or destination is part of a perfectly matched basic block, i.e, they satisfy the definition of a match of references, as described in §4.2. Additionally, we match the global offset table (*.got*), procedure linkage table (*.plt*), and any available exported symbols, if present, to obtain more matched references. Note that in practice, as well as in our experiments, binaries are generally stripped.

The system ends this phase with a list of matched references and a list of basic blocks that need to be reassembled.

5.4. Local Reassembly

Before beginning reassembly, we reserve space for the patch by extending the first load segment, or, if necessary, adding a new section to the vulnerable binary using lief. Although we could arbitrarily enlarge the segment, it is crucial to avoid disrupting the binary's loading memory layout. For instance, jumps into the *.got* are implemented as relative jumps. Therefore, we must ensure that relative distance of the loading addresses of the *.got* and the code during execution is not altered by our extension of the load segment. Typically, the space created is sufficient to accommodate the patch; however, a check is implemented to add a new section if it is not. This also ensures that subsequent re-patching of the same binary would not run out of available space.

Moving forward, the system starts to reassemble the identified basic blocks. The implementation closely follows Algorithm 1 to solve **C3** and **C4** but has to account for some technical details. Therefore, we will concentrate our exposition on the implementation of the essential functions.

The system iterates over the instructions of the patch in ascending address order checking if there is a reference originating from the instruction. The function `AdjustReference()` adjusts the reference such that there are no conflicts with possible shifts in later instructions. It checks whether the instruction reads or writes from a memory location that is part of the patch. If yes, it will reassemble the instruction but increase the distance to the memory location. The new memory location is saved.

The function `BackwardSlice()` implements a worklist algorithm over the DDG to find all statements in the intermediate representation VEX that use a register until its definition. Every register used in these statements is added to the worklist and the process is repeated to find their definitions. The result is a list of VEX statements.

The function `Solve()` processes a list of VEX statements, converting them into an equation system. This process involves iterating through the statements, transforming them into static single assignment form, and integrating them into the equation system. The final constraint added is the new target value. The function solves the equation system and all memory locations are returned with their defining instruction and the new values they require.

`LoadDataFrom()` is necessary to load the data from the destination of a reference that does not yet exist in the vulnerable binary. An example for that would be the string `"zero_length_keyword"` in Figure 2 at address `2400`. The function takes as input the address of the string in the patch binary and loads the bytes from this address until the first pair of consecutive null bytes. This heuristic works effectively for strings but may be insufficient for other data types. During our evaluation, we did not encounter the need for more sophisticated methods, but in principle we could leverage known work for type recovery and data structure reconstruction [42]–[44].

When the local reassembly algorithm is finished, i.e., all instructions have been reassembled, the system must account for shifts and some instructions might enlarge into 32-bit instructions during reassembling (**C5**). This will happen frequently where code is in Thumb mode, and the system has to encode far jumps. The prototype keeps track of those shifts and for every tracked shift it will add padding bytes after the shifted instruction so that the alignment of the following instructions is correct. Once all instructions are processed, the prototype changes all references affected by the shifts by the number of bytes that were added.

6. Evaluation

We now evaluate our approach on its ability to effectively patch real-world vulnerabilities in open-source libraries. In particular, we answer the following research questions:

- **RQ1**: Can Match&Mend successfully remove vulnerabilities from ELF binaries while preserving existing functionality, given a vulnerability, patch, proof of vulnerability, and test suite?
- **RQ2**: Can the system scale to support a wide range of IoT firmware images, rather than being limited to hand-crafted or case-specific scenarios?
- **RQ3**: What is the size overhead from patching vulnerabilities?

Answering the questions comes with two main challenges. First, we need a vulnerability and corresponding patch, an exploit, and a test suite for the firmware to demonstrate that the vulnerability is properly fixed and that existing functionality remains intact. Second, the evaluation must be scalable to ensure the system is not tailored to just one specific vulnerability or firmware. However, there is a lack of publicly available datasets that meet all these criteria. For this reason, we take a two-fold approach to evaluation.

All experiments were conducted using cross-compilation and emulation on a desktop machine running Ubuntu 22.04.4 LTS on an Intel Core i7-8700 CPU @ 3.20GHz. The patching process took 14 minutes on average; since the system is statically rewriting the binary, this process only needs to happen once.

6.1. RQ1 Patching Real-World Vulnerabilities

First, we demonstrate that the system can successfully patch real-world vulnerabilities in open-source libraries, using cases where a proof of vulnerability (PoV) and a corresponding test suite are available. To this end, we use the Magma dataset [45], which consists of open-source libraries with known vulnerabilities.

TABLE 1. RESULTS ON MAGMA.

CVE ID	Type	Library	O1	O2	O3
CVE-2018-13785	Integer overflow	LibPNG	✓	✓	✓
(Unspecified)	Memory leak	LibPNG	✓	✓	✓
CVE-2015-8781	OOB read	LibTIFF	✓	✓	✓
CVE-2016-9533	OOB read	LibTIFF	✓	✓	✓
CVE-2019-7663	0-pointer dereference	LibTIFF	✗	✗	✗
CVE-2017-11613	Resource exhaustion	LibTIFF	✗	✗	✗
CVE-2017-0663	Type confusion	LibXML2	✓	✓	✓
CVE-2017-7375	XML external entity	LibXML2	✗	✓	✓
CVE-2017-9048	Stack buffer overflow	LibXML2	✓	✓	✓
CVE-2015-8317	OOB read	LibXML2	✓	✓	✓
CVE-2016-1762	Heap buffer overread	LibXML2	✗	✓	✓
CVE-2016-6309	Use-after-free	OpenSSL	✓	✗	✗
CVE-2017-3735	OOB read	OpenSSL	✓	✓	✓
CVE-2016-6302	OOB read	OpenSSL	✓	✓	✓

Methodology. The libraries in the Magma dataset can be compiled with or without vulnerabilities, i.e., for each vulnerability we have (i) a version that represents the vulnerable library obtained from a firmware image, and (ii) a patched version compiled with the provided patch already applied. Since we want to simulate a real-world scenario, we compile each vulnerability with three different compilation settings, O1, O2, and O3, while the patch is always compiled with O2 and debug information. The two versions for each compilation setting are given to our prototype. For each vulnerability, the prototype analyzes, matches and locally reassembles the patch in the vulnerable binary, which results in a fixed version of the library. This fixed version is then tested to validate that the patch was successful.

We test the fixed binary to check (i) the proof of vulnerability and (ii) correctness. For (i), if there is a PoV available, we verify that the vulnerability no longer executes. Canaries inserted into the vulnerable libraries by Magma indicate whether a given vulnerability is reached and triggered. This allows us to test the patched binary by executing the library with the AFL++ [46] fuzzing harness from Magma and the proof of vulnerability as input. We consider the PoV test to be successful if the canary indicates that the vulnerability was not triggered. For correctness (ii), we run each fixed binary against its open-source test suite to validate that the patching has not rendered the library unusable. We consider the tests successful if the fixed binary passes the same tests as the vulnerable binary does, except for test cases it failed because of the vulnerability.

We consider a binary to be successfully patched if both the PoV test (i) and the test suite (ii) pass. We validated the soundness of our testing methodology by manually inspecting a sample of patched binaries with a disassembler. Recent work has shown how to validate patches at the binary level using symbolic execution [47], which could help to further automate validation checks.

Results. The Magma dataset contains six open-source libraries and one executable with known vulnerabilities. Since our focus lies on open-source libraries for ARM-based IoT-devices, we only consider the libraries in the dataset that we could successfully compile for this architecture with the correct configurations, i.e., LibPNG, LibTIFF, LibXML2, OpenSSL, and SQLite. Since there is no publicly available binary-level test suite for SQLite, and we did not receive a reply from developers upon request for an academic license, we excluded SQLite. Out of those libraries we selected all 14 vulnerabilities that are limited to one function and where MAGMA provided a POV. The resulting set covers a broad variety of vulnerability types, as shown in Table 1. All libraries were compiled using the gcc cross-compilation toolchain `arm-linux-gnueabihf-gcc`.

The adapted MAGMA dataset contains 14 libraries with different vulnerabilities, each compiled in three versions, resulting in 42 vulnerable libraries in total. Our prototype was able to process all of these. The detailed results are shown in Table 1. Out of the 42 libraries, 32 (76%) have been patched successfully; for the remaining 10, patching was unsuccessful. Two of these were not patched correctly because their optimization level (O1) led to wrong matches. Similarly, matching failed on levels O2 and O3 in one vulnerability in OpenSSL. The other six cases involve two vulnerabilities in LibTIFF. For each vulnerability, patching failed across all optimization levels, because the backward slicing was unable to correctly follow data flow through the stack in the patched version.

The limitations in function matching indicate that replacing the BinDiff implementation with more accurate machine-learning based systems could further improve the end-to-end performance of our system.

Despite limitations from BinDiff and fundamental challenges in static analysis of binaries, Match&Mend successfully patches the overwhelming majority of cases, demonstrating that it could improve security in practice.

6.2. RQ2 Patching IoT Firmware Images

The second part of the evaluation focuses on testing Match&Mend in a realistic deployment setting. Using firmware images from the Karonte dataset [48], we demonstrate that the prototype can operate effectively in real-world conditions. The dataset does not provide information about known vulnerabilities, let alone proofs of vulnerabilities, so we rely on public CVE databases for identifying candidate vulnerabilities and test suites for validating functionality. In combination with the evaluation of RQ1, this demonstrates that our approach is effective across a broad range of real-world firmware scenarios.

Methodology. The Karonte firmware dataset consists of 49 Linux-based firmware images from four major IoT vendors that make the firmware of their devices available for download: NETGEAR, TP-Link, D-Link, and Tenda. Since the dataset neither provides information about included open-source libraries nor known n-day vulnerabilities, we use the CVE Binary Tool [15] to identify open-source libraries and their versions in the firmware images. The CVE Binary Tool is a Python-based open source scanning tool maintained by Intel, which tries to automatically identify libraries and other types of software, as well as their specific version, and then matches this information with published CVE information. The identification mainly relies on relevant product and version strings in the scanned file. We excluded all files from the list of supposedly vulnerable files that were not compiled for a 32-bit ARM architecture. Furthermore, we selected six CVEs based on their relevance, i.e., the number of affected firmware images in the Karonte dataset, and on the ease of cross-compiling the corresponding libraries (libpcap, zlib, libpng, libflac) in all affected versions.

We built a script to prepare patched versions of the selected test subjects. Where possible, we automatically identified the next unaffected version for a given CVE, fetched the source code, and compiled it using a toolchain built with `buildroot` [49]. The toolchain uses either glibc [50] or uClibc-ng [51], depending on the loader in the firmware of the affected binary. In the case of CVE-2019-15165, the delta of the version fixing the CVE was too great, requiring manual backporting of the patch as if creating a hotfix for a previous source code release. Note that manual backporting hot patches at the source level is common practice in security-critical projects; because of its effectiveness for improving security, several research projects attempt to expand the scope of backporting by largely automating the process [52], [53].

Additionally, our script fetches and compiles the source code of the vulnerable library (i.e., the same version as in the firmware) to provide an environment to run the unit tests for the version that Match & Mend fixed. As we build the patched versions of binaries ourselves, we are free to choose compilation options. In particular, we can also build multiple versions with different compilation options to improve the quality of matching. For our experiments, we use the optimization levels Os (for size, common with embedded systems) and O2; for libpng, O2 yields best results, whereas Os is sufficient with all other targets. The process could be further improved using configuration inference, as demonstrated by OSSPatcher [11]. However, for our evaluation, we relied on lightweight matching, which was effective in our setting.

With these modifications, the CVE Binary Tool is able to supply the patching prototype with most of the information it needs to patch a vulnerable binary, i.e., the vulnerable binary itself, a compiled patched version, and a compiled affected version, as well as its source directory. The only missing information is the affected function, which we deduce from the CVE description and the source code of the affected version. Since CVE descriptions can be unclear or may lack important information [54], this is the most time-consuming part of the process. For example, a statement like "all versions before *1.30.9*" can be interpreted in multiple ways, such as referring to versions from *1.30.0* to *1.30.9*, or from *1.0.0* to *1.30.9*. Moreover, CVEs tend to describe the impact of a vulnerability rather than its root cause. Since

TABLE 2. RESULTS ON REAL WORLD FIRMWARE VULNERABILITIES. "AFFECTED" ARE DISTINCT BINARIES IN DIFFERENT FIRMWARE IMAGES.

CVE ID	Library	Affected	Patched	Passed	Success
CVE-2019-15165	libpcap	7	3	3	43%
CVE-2016-9840	zlib	33	33	33	100%
CVE-2016-9842	zlib	24	24	24	100%
CVE-2016-9843	zlib	33	33	33	100%
CVE-2016-10087	libpng	7	6	6	85%
CVE-2020-22219	libflac	27	27	27	100%
Total		131	126	126	96%

Match&Mend is designed to patch n-day vulnerabilities, we generally rely on the assumption that the affected function is known. While CVE entries are the most obvious source of this information, bug reports or commit messags are equally valid sources. Fortunately, the importance of vulnerability databases has become increasingly clear to vendors and policy makers, and we expect the quality of such reports to improve. Overall, we were able to identify the affected function for six CVEs in different versions of different libraries, which we then patched and tested.

To test a patched library, we cross-compile the test suite for the affected library, if available. If no test suite exists or if it cannot be cross-compiled, we write manual test cases for the affected functions. All tests are executed within the filesystem of the original firmware image. Although this does not allow us to directly confirm that a vulnerability has been fixed, we know after testing that a patch was applied and that the patched library remains functional.

Results. Our system can successfully patch 126 out of 131 vulnerabilities across all 30 ARM32 firmware images in the Karonte dataset. As shown in Table 2, whenever the system was able to apply a patch, the corresponding library also passed its test suite. In the five remaining cases, Match&Mend was unable to patch the vulnerability because the underlying analysis tool, angr, failed to construct an interprocedural control flow graph for the affected library. These results indicate that our approach is effective in practice even when dealing with libraries that are stripped of section headers and symbols and compiled with unknown configurations. Overall, Match&Mend was able to patch all ARM32 firmware images in the Karonte dataset.

6.3. RQ3 Size Overhead

Another important aspect of the performance of the system is its ability to adhere to the micro-patching strategy, i.e., to patch only the affected functions and not the entire library. The design of Match&Mend is based on basic block matching, enabling the system to patch individual basic blocks to remove vulnerabilities. In practice, however, the system often needs to patch more than a few basic blocks either due to matching difficulties or because the patch itself spans multiple blocks. Often, the system would end up patching entire functions.

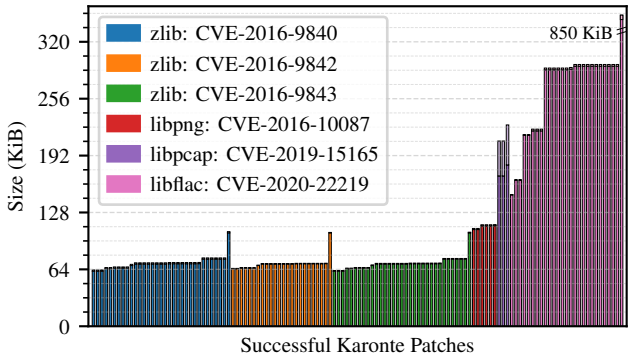

Figure 4. Histogram showing the size of each vulnerable library and the additional size of its patch.

Nevertheless, the analysis is identical whether we use basic blocks or functions as the fundamental units for matching. Using basic blocks provides us with the opportunity to patch very small units whenever possible, but can be extended seamlessly for larger areas of the binary. Figure 4 shows the sizes of the vulnerable binaries and the bytes needed to patch them. The observed sizes of the identified patches support the assumption that security patches tend to be small, supporting the micro-patching strategy ranging from one to 34 lines of code. While the current prototype does not yet optimize the use of available space and always adds the patch segment at a fixed size, the average patch sizes observed in real-world firmware images range from 5 to 46 KiB or 9 to 66 basic blocks, compared to library sizes between 65 and 850 KiB. The larger patches correspond to cases where larger amounts of data had to be copied; the only outlier is a library that includes debug information, which is uncommon with production firmware.

As the patches are added to a new section of the binary, if required, we can in principle patch an arbitrary number of vulnerabilities over time. While theoretically the file size could grow larger than available memory, the patch sizes shown in Figure 4 indicate that this is unlikely to occur within the lifespan of a device.

Thus, we can conclude that Match&Mend achieves the goal of micro-patching, by only modifying a very small fraction of the code in the affected libraries. This presents a significant improvement over replacing entire libraries with fixed versions, which is error-prone due to the wide variety of compiler options, build configurations, and potential custom adaptations.

7. Limitations

In the following, we discuss some limitations of the implementation and evaluation of our system.

The accuracy of our reassembling process depends on the quality of the generated control flow graph (CFG) and data flow graph (DFG). In general, the current implementation strikes a reasonable balance between precision and performance, but it has some limitations. For instance, data flows that pass through the stack are not yet tracked reliably.

Since our use case allows partial access to source-level information and limiting the scope locally, this context could be leveraged to improve the precision of CFG and DFG generation in targeted areas.

Identifying the affected functions in the source code is a prerequisite for our approach; however, locating the corresponding functions in the vulnerable binary presents a separate challenge. As this is not the main focus of our prototype, we opted for a simple and practical solution. Searching for symbol names is a reasonable first step, but it fails when the binary is stripped, a common scenario, particularly in embedded systems. In such cases, we fall back on a BinDiff based implementation using angr, which performs well in many cases. Nevertheless, matching functions becomes difficult when the target has been inlined or compiled with significantly different settings. While our current method works in most cases, it is limited in handling these edge scenarios. The field of binary code similarity analysis offers many improvements [36], and integrating machine learning models tailored to this task could enhance locating the function.

Local Reassembly, as implemented in Match&Mend, integrates code in binary form into an existing binary. To match substantial parts of the patch binary to the vulnerable binary, it is beneficial to approximate the original compilation settings as closely as possible. This observation is also supported by our MAGMA evaluation, where several failures were caused by incorrect function matches across different optimization levels. We do not require a fully exact reconstruction of the build environment, which would only be necessary when rebuilding an entire library, but recovering key compiler parameters has a strong impact on matching quality. One particularly influential parameter is the compiler optimization level. Our current approach is pragmatic: we compile the patched source using several candidate optimization levels, typically four or five, and then select the configuration that yields the highest similarity to the original binary. This procedure mitigates limitations in our function matching, because a binary compiled with an appropriate optimization level more closely resembles the original binary

8. Related Work

Now we discuss existing work for static binary rewriting and binary patching, as shown in Table 3, and whether they offer a solution for the identified challenges **C1** to **C5**. To the best of our knowledge, Match & Mend is the only tool that solves all challenges for Linux-based ARM32 binaries.

Automated binary patching of vulnerabilities requires precise modifications to address specific security issues, often replacing or fixing faulty logic. Existing techniques for binary rewriting merely shift the problem of patching to a different level by creating modifiable intermediate representations or reassembleable assembly while not providing a technique for systematically integrating patches into the binary. Most work is focused on rewriting binaries for analysis or monitoring purposes. While the core challenges of binary rewriting remain similar independent of the rewriting goal, techniques for instrumentation are not necessarily suitable for patching vulnerabilities. The key difference lies in the type of binary code added by the binary rewriting process. Instrumentation, for example, refers to adding code for dynamic analysis, usually in a way that does not alter the intended functionality of the original program. Still, the core challenges of binary analysis like control flow recovery, data flow analysis or symbolization remain and do not change with the goal of the binary rewriting. Table 3 provides an overview of binary rewriting and patching tools, their focus and target architecture and whether they address the challenges with a full solution (●), a partial solution (◑), or a solution specific to the dynamic setting ($●^d$). In the following we will discuss the work in more detail.

Overall, we can classify all approaches into two categories based on their impact on the binary. The first category is global rewriting, where the entire binary is transformed. The second category is local rewriting, which focuses on modifying only specific parts of the binary. Match&Mend follows a local micro-patching strategy.

8.1. Global Rewriters

Tools in the first category are commonly designed to instrument binaries for purposes such as fuzzing, hardening, or analysis. While they modify the entire binary and are not specifically tailored for patching vulnerabilities, their binary rewriting capabilities could, in principle, be adapted to our use case. However, their global modifications conflict with our minimally invasive and localized approach. In the following, we analyze how these tools address challenges **C1** to **C5**.

RetroWrite [55] and **Armore** [59] create reassembleable assembly through symbolization for 64-bit x86 and AArch64 PIE binaries, respectively. Leveraging relocation information of position-independent binaries, they do not rely on heuristics to symbolize the binary. Since their focus lies more on instrumentation and not on patching known vulnerabilities, they do not offer solutions for patch or vulnerability identification (**C1** and **C2**). While the authors show that Armore can be used to patch vulnerabilities, doing so requires an expert who manually patches the reassembleable assembly. Hence, there is no approach to integrate control or data flow of a patch into the binary. Encoding modified instructions is solved implicitly by RetroWrite and specifically Armore by symbolizing the entire binary. Both tools work on Linux-based systems but are not compatible with ARM32 binaries.

Repica [56] is a static binary instrumentation technique that symbolizes the entire binary. Using a value-set-analysis specialized for position independent ARM binaries, Repica identifies and processes relative addresses. Any changes to be made need to be specified manually in an instrumentation specification, shifting the problems of locally integrating a patch to another level and hence not addressing challenges **C1** to **C4**. Challenge **C5**, encoding modified instruction is

TABLE 3. COMPARISON OF DIFFERENT BINARY REWRITING AND PATCHING TOOLS.

Name	Rewriting	Architectures	Modification	Main Goal	C1	C2	C3	C4	C5
RetroWrite [55]	Static	x86_64	Global	Instrumentation	○	○	○	○	●
Repica [56]	Static	ARM32, AArch64	Global	Instrumentation	○	○	○	○	●
Egalito [18]	Static	x86_64, AArch64	Global	Instrumentation	○	○	○	○	●
DDisasm [57]	Static	x86_64	Global	Disassembly	○	○	○	○	●
ICFGP [58]	Static	x86_64, ppc64le, AArch64	Local	Instrumentation	○	○	○	○	○
E9Patch [17]	Static	x86_64	Local	Instrumentation	○	○	○	○	○
MultiVerse [19]	Static	x86_32	Global	Instrumentation	○	○	○	○	●
Armore [59]	Static	AArch64	Global	Instrumentation	○	○	○	○	●
RevARM [16]	Static	ARM32	Local	Instrumentation	○	○	○	○	●
BinPatch [60]	Static	x86_64	Local	Patching	◑	◑	◑	◑	○
OSSpatcher [11]	Dynamic	ARM32, AArch64, x86_64	Local	Patching	●	●	●[d]	●[d]	○
Match & Mend	Static	ARM32	Local	Patching	●	●	●	●	●

also implicitly solved by symbolizing the entire binary. The tool is compatible with ARM32 and AArch64 binaries.

DDisasm [57] is a static binary rewriter, developed in Datalog for 32-bit x86 binaries, and since publication, also ARM32. It statically recovers control- and data flow information and uses reassembly heuristics optimized through the use of datalog. The entire binary is symbolized focusing on analysis and monitoring purposes rather then automated patching. Challenges **C1** to **C4** are not addressed by DDisasm but symbolizing the entire binary implicitly solves challenge **C5**.

Egalito [18] recompiles 64-bit binaries (x86 or ARM) for rewriting by first lifting them to an intermediate representation. This global approach shifts the challenges **C1** to **C4** to the level of the intermediate presentation but does not solve them. Similar to the other global approaches the encoding of modified instructions is implicitly solved by the recompilation process.

MultiVerse [19] statically rewrites x86 binaries without heuristics. It completely disassembles instruction sequences starting from every byte offset in the binary's text section constructing a set of instructions that contains a subset with all valid instructions. Indirect control flow is rewritten using a table mapping new addresses to the original ones. MultiVerse is then able to relocate all instructions to any other location without changing the original control flow. The main purpose of their technique is disassembly and instrumentation and does not offer a solution for **C1** to **C4**. For x86 binaries they solve challenge **C5**, but this approach is not applicable for ARM architectures.

8.2. Local Rewriters

The second category of tools focuses on local rewriting and includes our own tool, Match & Mend.

ICFGP [58] rewrites 64-bit binaries (x86_64, AArch64, ppc64le) locally by inserting trampolines. The tool is implemented on top of Dyninst [61] and implements an analysis to find the best places to put trampolines. It uses the trampolines to jump to a specially crafted instrumentation area and then incrementally tries to reduce the amount of trampolines while changing all necessary control flow so that the original semantics of the binary can still be executed. The purpose of their tool is instrumentation. In their evaluation they use empty instrumentation to show that their tool does not break the binary. Challenge **C1** to **C5** are not addressed.

E9Patch [17] follows a minimally invasive local approach by inserting trampolines into x86_64 binaries so that binary code can be injected into it. It works without recovering control flow and on a very low level. As input, it takes an unpatched binary, disassembly information, a set of patch instruction locations and trampoline templates. This input needs to be provided by a user and does not offer an automated solution for challenges **C1** to **C5**.

RevARM [16] instruments ARM (both ARM32 and AArch64) binaries without limitation on the target platform. It takes an input binary and instrumentation specifications such as security policies to enforce and target embedding/extraction locations and generates an instrumented binary. The instrumentation specification requires information about the binary, the vulnerability and a possible binary code fix that is not easily available for IoT-firmware. Our system tries to automate the process of creating this information and therefore does not require a manually created instrumentation specification. Thus challenges **C1** to **C4** are not addressed, but it solves challenge **C5** by inserting NOP statements to keep alignment and adjusting the encoding of references when necessary.

BinPatch [60] is an automated binary patching tool for IA32 binaries. Similar to our system the tool uses binary code similarity analysis together with a description of the vulnerability to identify patch and vulnerability (challenges **C1** and **C2**). Additionally, BinPatch requires a known exploit as a trigger for the vulnerability. It uses this exploit as an input case for symbolic execution. BinPatch executes a self-compiled version of the patch and symbolically executes the vulnerable firmware version with this input. Afterwards, it compares the results of the two executions and uses the gained information to rewrite the vulnerable function. This approach limits the applicability of BinPatch to vulnerabilities that have a known exploit. As we have seen, even for a curated dataset like Magma this is the case for only about 45% of the vulnerabilities. Furthermore, BinPatch is not able to patch vulnerabilities that have the same CFG as their patch and is limited to vulnerabilities that are patched by adding new conditions or modifying the original ones. It

can only perform the patching if the data structure of the vulnerable and the patch function is the same. Under those constraints on control flow and data flow, BinPatch solves challenges **C3** and **C4**. Challenge **C5** is also solved for the targeted IA32 architecture.

OSSPatcher [11] is the most closely related work to ours, as shown in Table 3, and is capable of dynamically patching known vulnerabilities in binaries on Android. Like Match&Mend, OSSPatcher leverages information from publicly available source code to guide its binary patching process. It identifies both the patch and the vulnerable code using CVE information and function-level matching, thereby addressing challenges **C1** (on the source level) and **C2**. Once a source-level patch is identified, OSSPatcher performs a feasibility analysis to determine whether the patch can be applied. To generate a patch, OSSPatcher compiles a stub library and a patch library, with dependencies to both the original vulnerable libraries and the stub. Instead of employing static binary rewriting to solve challenges **C3** (Control Flow Integration) and **C4** (Data Flow Integration) this design delegates the resolution of references to the linker and loader at runtime. To deploy patches, OSSPatcher uses `ptrace` to monitor Android's Zygote process and injects the libraries in memory on matching `open` and `mmap` system calls. While this is an elegant design within the lifecycle of Android applications, it is unsuitable for our use case. A similar deployment mechanism for general embedded Linux distributions would require (binary) patches to the kernel, which goes against our premise of minimizing possibly damaging side effects. Furthermore, the creation of separate libraries causes significant space increases per patch, with the evaluation reporting up to 80 KB. While OSSPatcher is not publicly available for direct comparison, this would, in our dataset, amount to a size overhead of up to 125% per patch for the smallest library.

The authors of OSSPatcher acknowledge that static binary rewriting could be an alternative implementation path. However, pursuing this would necessitate addressing challenges **C3** (Control Flow Integration) and **C4** (Data Flow Integration) differently, as such information can only be resolved dynamically by the loader. For static binary rewriting, OSSPatcher does not provide mechanisms to solve these challenges, underscoring the need for a dedicated solution such as Match&Mend, which is explicitly designed to address them in the static binary rewriting context.

9. Conclusion

In this work, we were able to show that minimally invasive local reassembly can be a viable solution for patching (n-day) vulnerabilities in accessible IoT firmware, without vendor support. To this end, we implemented Match & Mend, a prototype that automatically reassembles patches for known vulnerabilities in ARM 32-bit firmware binaries. The system is based on a novel three-phase approach that includes binary analysis, matching and local reassembly. It implements a micro-patching strategy that utilizes a locally scoped symbolic mapping.

The evaluation of our approach on known vulnerabilities and real world firmware images highlights the practicality and potential of binary-level patching. By focusing on minimal invasiveness and reducing the risk of breaking changes, our system achieves a high success rate at minimal size overhead. Moreover, our approach will benefit from further advances in binary code similarity detection, automated backporting, and patch verification, which are all active areas of research; therefore, we consider minimally invasive local reassembly a promising approach towards enhancing the security and extending the lifespan of IoT devices.

Acknowledgment

This work was supported by the Bavarian State Ministry of Science and Arts (BayStMWK), under project "ForDaySec: Security in Everyday Use of Digital Technologies (fordaysec.de)" of the Bavarian Research Association.

References

[1] Linux Foundation, "The Linux foundation announces Yocto project steering group and release 1.0," https://www.linuxfoundation.org/press/press-release/the-linux-foundation-announces-yocto-project-steering-group-and-release-1-0, 2011.

[2] N. Nino, R. Lu, W. Zhou, K. H. Lee, Z. Zhao, and L. Guan, "Unveiling IoT security in reality: A firmware-centric journey," in *33rd USENIX Security Symp.*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024.

[3] D. Kumar, K. Shen, B. Case, D. Garg, G. Alperovich, D. Kuznetsov, R. Gupta, and Z. Durumeric, "All things considered: An analysis of IoT devices on home networks," in *28th USENIX Security Symp.* USENIX Association, 2019, pp. 1169–1185.

[4] European Parliament and Council of the European Union, "Regulation (EU) 2024/2847 on horizontal cybersecurity requirements for products with digital elements," Official Journal of the European Union, L 2024/2847, Oct. 2024, http://data.europa.eu/eli/reg/2024/2847/oj.

[5] S. Vasile, D. F. Oswald, and T. Chothia, "Breaking all the things – A systematic survey of firmware extraction techniques for IoT devices," in *17th Int. Conf. Smart Card Research and Advanced Applications (CARDIS)*, ser. LNCS, vol. 11389. Springer, 2018, pp. 171–185.

[6] M. Ibrahim, A. Continella, and A. Bianchi, "AoT – Attack on things: A security analysis of IoT firmware updates," in *8th IEEE European Symp. on Security and Privacy, (EuroS&P)*. IEEE, 2023, pp. 1047–1064.

[7] Y. Wu, J. Wang, Y. Wang, S. Zhai, Z. Li, Y. He, K. Sun, Q. Li, and N. Zhang, "Your firmware has arrived: A study of firmware update vulnerabilities," in *33rd USENIX Security Symp.*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024.

[8] Y. Su and D. C. Ranasinghe, "Leaving your things unattended is no joke! memory bus snooping and open debug interface exploits," in *IEEE International Conf. on Pervasive Comp. and Comm. Workshops (PerCom)*. IEEE, 2022, pp. 643–648.

[9] J. Vijayan, "Most IoT hardware dangerously easy to crack," 2020, accessed: 2025-05-26. [Online]. Available: https://www.darkreading.com/iot/most-iot-hardware-dangerously-easy-to-crack

[10] roleoroleo, "yi-hack-allwinner-v2," 2024, accessed: 2025-05-26. [Online]. Available: https://github.com/roleoroleo/yi-hack-Allwinner-v2

[11] R. Duan, A. Bijlani, Y. Ji, O. Alrawi, Y. Xiong, M. Ike, B. Saltaformaggio, and W. Lee, "Automating patching of vulnerable open-source software versions in application binaries," in *Network and Distributed System Security Symp. (NDSS)*. The Internet Society, 2019.

[12] W. Tang, Y. Wang, H. Zhang, S. Han, P. Luo, and D. Zhang, "Libdb: An effective and efficient framework for detecting third-party libraries in binaries," in *19th IEEE/ACM Int. Conf. Mining Software Repositories (MSR)*. ACM, 2022, pp. 423–434.

[13] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying open-source license violation and 1-day security risk at large scale," in *Proc. ACM SIGSAC Conf. on Computer and Communications Security, CCS*, B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 2169–2185.

[14] Y. David, N. Partush, and E. Yahav, "Firmup: Precise static detection of common vulnerabilities in firmware," in *Proc. 23rd Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2018, pp. 392–404.

[15] Intel, "CVE binary tool," 2025, accessed: 2025-05-14. [Online]. Available: https://github.com/intel/cve-bin-tool

[16] T. Kim, C. H. Kim, H. Choi, Y. Kwon, B. Saltaformaggio, X. Zhang, and D. Xu, "Revarm: A platform-agnostic ARM binary rewriter for security applications," in *Proc. Computer Security Applications Conf. (ACSAC)*. ACM, 2017, pp. 412–424.

[17] G. J. Duck, X. Gao, and A. Roychoudhury, "Binary rewriting without control flow recovery," in *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2020, pp. 151–163.

[18] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis, "Egalito: Layout-agnostic binary recompilation," in *Proc. 25th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2020, pp. 133–147.

[19] E. Bauman, Z. Lin, and K. W. Hamlen, "Superset disassembly: Statically rewriting x86 binaries without heuristics," in *Network and Distributed System Security Symp. NDSS*. The Internet Society, 2018.

[20] Y. He, Z. Zou, K. Sun, Z. Liu, K. Xu, Q. Wang, C. Shen, Z. Wang, and Q. Li, "Rapidpatch: Firmware hotpatching for real-time embedded devices," in *31st USENIX Security Symp.*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 2225–2242.

[21] C. Niesler, S. Surminski, and L. Davi, "HERA: hotpatching of embedded real-time applications," in *Network and Distributed System Security Symp., NDSS*. The Internet Society, 2021.

[22] M. Salehi and K. Pattabiraman, "Autopatch: Automated generation of hotpatches for real-time embedded devices," in *Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS)*, B. Luo, X. Liao, J. Xu, E. Kirda, and D. Lie, Eds. ACM, 2024, pp. 2370–2384.

[23] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. R. Weippl, "From hack to elaborate technique - A survey on binary rewriting," *ACM Comput. Surv.*, vol. 52, no. 3, pp. 49:1–49:37, 2019.

[24] M. Jiang, Y. Zhou, X. Luo, R. Wang, Y. Liu, and K. Ren, "An empirical study on ARM disassembly tools," in *Proc. Int. Symp. Software Testing and Analysis (ISSTA)*, S. Khurshid and C. S. Pasareanu, Eds. ACM, 2020, pp. 401–414.

[25] Y. Ye, Z. Zhang, Q. Shi, Y. Aafer, and X. Zhang, "D-ARM: disassembling ARM binaries by lightweight superset instruction interpretation and graph modeling," in *Proc. IEEE Symp. Security and Privacy (S&P)*. IEEE, 2023, pp. 2391–2408.

[26] H. Theiling, "Extracting safe and precise control flow from binaries," in *7th Int. Workshop Real-Time Computing and Applications Symp. (RTCSA)*. IEEE Computer Society, 2000, pp. 23–30.

[27] J. Kinder, H. Veith, and F. Zuleger, "An abstract interpretation-based framework for control flow reconstruction from binaries," in *Proc. 10th Int. Conf. Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, vol. 5403. Springer, 2009, pp. 214–228.

[28] J. Kinder and D. Kravchenko, "Alternating control flow reconstruction," in *Proc. 13th Int. Conf. Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, vol. 7148. Springer, 2012, pp. 267–282.

[29] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," in *Proc. IEEE Symp. Security and Privacy (S&P)*. IEEE, 2021, pp. 833–851.

[30] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, "SOK: (state of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Security and Privacy (S&P)*, 2016, pp. 138–157.

[31] Shellphish, "patcherex," 2023, accessed: 2023-12-27. [Online]. Available: https://github.com/angr/patcherex

[32] R. Thomas, "Lief – library to instrument executable formats," apr 2017, accessed: 2023-12-27. [Online]. Available: https://lief.quarkslab.com/

[33] H. Flake, "Structural comparison of executable objects," in *Proc. GI SIG SIDAR Workshop Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, ser. LNI, vol. 46. GI, 2004, pp. 161–173.

[34] A. Jia, M. Fan, W. Jin, X. Xu, Z. Zhou, Q. Tang, S. Nie, S. Wu, and T. Liu, "1-to-1 or 1-to-n? investigating the effect of function inlining on binary similarity analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, May 2023. [Online]. Available: https://doi.org/10.1145/3561385

[35] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *IEEE Symp. Security and Privacy (S&P)*. IEEE, 2019, pp. 472–489.

[36] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," in *Annu. Network and Distributed System Security Symp. (NDSS)*. The Internet Society, 2020.

[37] I. U. Haq and J. Caballero, "A survey of binary code similarity," *ACM Comput. Surv.*, vol. 54, no. 3, pp. 51:1–51:38, 2022.

[38] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, "Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned," *IEEE Trans. Software Eng.*, 2022.

[39] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, "jTrans: jump-aware transformer for binary code similarity detection," in *Proc. ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA)*. ACM, 2022.

[40] H. Wang, Z. Gao, C. Zhang, Z. Sha, M. Sun, Y. Zhou, W. Zhu, W. Sun, H. Qiu, and X. Xiao, "CLAP: learning transferable binary code representations with natural language supervision," in *Proc. ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA)*. ACM, 2024, pp. 503–515.

[41] M. Dannehl, S. Valenzuela, and J. Kinder, "Which instructions matter the most: A saliency analysis of binary function embedding models," in *Proc. IEEE Symp. Security and Privacy Workshops (SPW), Deep Learning Security and Privacy Workshop (DLSP)*. IEEE, 2025, pp. 145–151.

[42] J. Lee, T. Avgerinos, and D. Brumley, "TIE: principled reverse engineering of types in binary programs," in *Proc. Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2011.

[43] M. Noonan, A. Loginov, and D. R. Cok, "Polymorphic type inference for machine code," in *Proc. 37th ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*. ACM, 2016, pp. 27–41.

[44] Y. Wang, R. Liang, Y. Li, P. Hu, K. Chen, and B. Zhang, "TypeForge: Synthesizing and selecting best-fit composite data types for stripped binaries," in *Proc. IEEE Symp. Security and Privacy (S&P)*. IEEE, 2025, pp. 3070–3087.

[45] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *CoRR*, vol. abs/2009.01120, 2020. [Online]. Available: https://arxiv.org/abs/2009.01120

[46] A. Fioraldi, D. C. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies, (WOOT)*. USENIX Association, 2020.

[47] H. Wu, J. Wu, R. Wu, A. Sharma, A. Machiry, and A. Bianchi, "Veribin: Adaptive verification of patches at the binary level," in *Network and Distributed System Security Symp. (NDSS)*. The Internet Society, 2025.

[48] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Karonte: Detecting insecure multi-binary interactions in embedded firmware," in *IEEE Symposium on Security and Privacy, (SP)*. IEEE, 2020.

[49] Buildroot Developers, "Buildroot," 2025, accessed: 2025-05-14. [Online]. Available: https://buildroot.org

[50] GNU Project, "glibc," 2025, accessed: 2025-05-14. [Online]. Available: https://www.gnu.org/software/libc/

[51] W. Brodkorb, "uclibc-ng," 2025, accessed: 2025-05-14. [Online]. Available: https://www.uclibc-ng.org

[52] R. Shariffdeen, X. Gao, G. J. Duck, S. H. Tan, J. Lawall, and A. Roychoudhury, "Automated patch backporting in linux (experience paper)," in *Proc. Int. Symp. Software Testing and Analysis (ISSTA)*. ACM, 2021, pp. 633–645.

[53] S. Yang, Y. Xiao, Z. Xu, C. Sun, C. Ji, and Y. Zhang, "Enhancing OSS patch backporting with semantics," in *Proc. ACM SIGSAC Conf. Computer and Commun. Security (CCS)*. ACM, 2023, pp. 2366–2380.

[54] P. Kuehn, M. Bayer, M. Wendelborn, and C. Reuter, "OVANA: an approach to analyze and improve the information quality of vulnerability databases," in *Int. Conf. Availability, Reliability and Security (ARES)*. ACM, 2021, pp. 22:1–22:11.

[55] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting COTS binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy, (SP)*. IEEE, 2020.

[56] D. Ha, W. Jin, and H. Oh, "REPICA: rewriting position independent code of ARM," *IEEE Access*, vol. 6, 2018.

[57] A. Flores-Montoya and E. M. Schulte, "Datalog disassembly," in *29th USENIX Security Symp.* USENIX Association, 2020.

[58] X. Meng and W. Liu, "Incremental CFG patching for binary rewriting," in *26th ACM Intern. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2021.

[59] L. D. Bartolomeo, H. Moghaddas, and M. Payer, "Armore: Pushing love back into binaries," in *32nd USENIX Security Symp.* USENIX Association, 2023, pp. 6311–6328.

[60] Y. Hu, Y. Zhang, and D. Gu, "Automatically patching vulnerabilities of binary programs via code transfer from correct versions," *IEEE Access*, vol. 7, pp. 28 170–28 184, 2019.

[61] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *Proc. of ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools (PASTE'11)*, J. Foster and L. L. Pollock, Eds. ACM, 2011, pp. 9–16.