IRENE: A Toolchain for High-level Micropatching through Recompilable Sub-function Regions

Ian Smith
Trail of Bits
New York, United States
ian.smith@trailofbits.com

Francesco Bertolaccini
Trail of Bits
New York, United States
francesco.bertolaccini@trailofbits.com

William Tan
Trail of Bits
New York, United States
william.tan@trailofbits.com

Michael D. Brown
Trail of Bits
New York, United States
michael.brown@trailofbits.com

***Abstract*—Devices with long life-cycles extend the duration of a software product's exposure. Vulnerabilities in the device's software may be discovered after the toolchain for building software for that device has been deprecated, the source-code for the program that is running on the device is unavailable, or both. Current solutions for patching these latent vulnerabilities are insufficient, either requiring deep technical expertise and manual effort or producing a large set of changes to the target binary. The large number of changes in the target program makes validating the patch difficult.**

We present IRENE, a decompiler that produces recompilable decompilation for patching. IRENE decompiles sub-function regions of a binary program to separate recompilable regions of code. The IRENE compiler replaces the original binary region with recompiled assembly, representing the user's patch. IRENE allows developers to compose patches by modifying a high-level representation of the target program, but produces targeted micropatches. We evaluate the size of IRENE produced micropatches against patches produced both by recompilation and manual assembly patching in a case study. This evaluation shows a patch size reduction of 4x compared with recompilation and a 21x size reduction when compared with recompiling the patch with a newer toolchain version.

***Index Terms*—binary patching, recompilation, binary rewriting, binary analysis**

I. Introduction

Software targeting embedded devices and operational technologies (OT) are susceptible to a wide variety of vulnerabilities [1]. The software on these devices have longer life-cycles when compared with user applications. Unfortunately, these longer life-cycles lead to significant challenges when attempting to patch security critical bugs deployed on critical systems. The original source code may no longer be available or the toolchain that was used to produce the original binary may have been lost. Building a patch in these cases may require rebuilding the entire firmware package with a new toolchain or new version of the source code in order to deploy a small vulnerability fix to a device. Completely rebuilding the application results in a large change in program behavior, requiring retesting the entire device's functionality. These difficulties cause deployed mission critical systems with known vulnerabilities to remain unpatched in the field.

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions, and/or findings contained in this material are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for Public Release, Distribution Unlimited

Alternatively, engineers can apply patches to a representation of the binary. Static binary rewriting is a heavily explored topic with approaches including: decompilation and recompilation, detour-based binary patching, and binary disassembly and reassembly [2]–[4]. Of these approaches, only static decompilation and recompilation allows modification of the binary in a high-level language familiar to developers. Detoured-based patching and binary reassembly, require an understanding of the target binary to affect a desired patch, preventing software developers from using these techniques. On the other hand, full-binary decompilation and recompilation results in a new binary with potential changes in all functions of the binary, and requires a toolchain that can build the entire firmware package. We developed IRENE to achieve the best of both worlds by combining decompilation-based rewriting with detour-based patching. Users can replace a sub-function region of code with a detour to a patch expressed in a high-level C-like language. IRENE only changes the portion of the binary required to create the patch and can express patches at a high-level.

II. Related Work

Static binary disassembly and reassembly allows for modifying binaries at the assembly level in a layout agnostic way. New code can be inserted with references to old symbols [3], [4]. These techniques require complete disassembly and reference recovery in order to guarantee that all references are modified appropriately when code is relocated. Both disassembly and reference recovery require heuristics even in the presence of position-independent-code (PIC) [4]. For non-PIC firmware, performing symbolization accurately across the whole binary becomes intractable [3]. Using these tools for binary patching requires the user to modify a machine dependent assembly representation. Reassembly can result in a new layout that obfuscates the patched binary's relationship to the original binary, requiring substantial testing to validate the behavior of the patch.

On the other hand, detour-based patching can modify small portions of the binary directly, without reassembly or recompilation [2]. Detouring preserves the original layout of the binary, allowing the insertion of new code, and

modification of old code without control flow or reference recovery (except for references used by the patch). To achieve a patch that modifies an instruction or set of instructions, the relevant instructions are replaced with a jump to a location with free space. New or modified instructions are placed in this location along with a jump back to the instructions after the replaced instruction. E9Patch uses a variety of patching and detour techniques to ensure that a detour is inserted without affecting program counter relative instructions [2]. The applicability of detour-based patching to a wide variety of target binaries, while affecting a minimal portion of the input binary motivates the design of IRENE. IRENE extends detour-based patching with high-level decompilation of sub-function regions to achieve high-level micropatches.

III. Methodology

IRENE enables sub-function micropatching by soundly decomposing a function into binary regions that are decompiled independently. Throughout the lifting and lowering process IRENE maintains a close correspondence between a region of the binary, an LLVM representation of that region, and binary interface constraints on that LLVM representation [5]. These three components allow any lifted region to be modified, recompiled using the LLVM toolchain, and inserted into the binary, while preserving a simulation relation between the modified decompilation and the binary function.

A. Limitations of Approach

Sub-function decompilation induces some limitations in IRENE's patching approach. In order to recompile the patched region, an LLVM backend must be available for the target platform. IRENE is dependent on the underlying code recovery and disassembly algorithms used to build a program representation for a target sub-function region. In general, correct and accurate disassembly of a code region is undecidable [2]. A user may need to repair types, references, or control flow to retrieve an accurate recompilable region for code replacement.

B. Motivation for Binary Regions

IRENE's approach avoids key difficulties in soundly micropatching decompilation by ensuring that decompilation always has a direct relationship to binary semantics. Typical decompilers represent provenance between assembly instructions and decompilation through a multi-valued mapping from statement to influencing instructions. In the simplest case this provenance information can be rendered as a mapping from address to decompiled statement.

```
mov r3, r7 ; 0x0      r3=r7; // 0x0
ldr r5, [r6] ; 0x2    *r3=r4; // 0x4
str r4, [r3] ; 0x4    r5=*r6; // 0x2
```

Fig. 1. Decompilation provenance for example statements.

Figure 1 highlights decompilation of a simple basic block with a typical optimization that results in a reordering of the store instruction at `0x4` to before the load instruction at `0x2`. In the decompilation, we give each variable associated with a register the register's name for simplicity. The reordering of `r5=*r6` with `*r3=r4` is semantically valid if the decompiler determines that on entry to this block of assembly `r7` cannot alias `r6`. If this aliasing assertion holds, then the stored to address cannot affect the loaded value, so each representation is observationally equivalent.

```
mov r3, r7 ; 0x0      r3=r6; // 0x0
```

Fig. 2. Micropatch to the decompilation in Figure 1.

We now consider a micropatch highlighted in Figure 2. The user simply changes the assignment of `r3` to be equal to `r6`, thus the load from `r6` in `r5=*r6` is expected to load the value r4 that is stored in the statement `*r3=r4`. A naive patch localization strategy would simply change the instruction at `0x0` to `mov r3, r6`, but this compiled patch does not represent what the user saw in their modified decompilation. The load must occur after the store in order to have the user's intended effect, therefore all three instructions must be modified to reproduce the user's intended change. This example highlights the danger in relying on naive decompilation provenance for patch compilation.

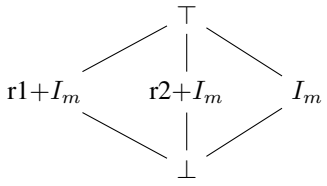
IRENE resolves this issue by making all decompilation localizable by construction. A function is decomposed into a set of control-flow-free, patcheable regions. Regions are optimized independently, allowing internal rewriting and reordering, but retaining the region's relationship to the binary location it represents. Optimization between regions only occurs based on explicit binary facts that are inferred during analysis and enforced as contracts between regions. For the reordering shown in Figure 1 to occur, all three instructions must be in the same region. IRENE therefore replaces the range of instructions `0x0` to `0x4` with recompiled code from Figure 2, guaranteeing that the patched binary preserves the behavior that the user saw in their modified decompilation.

C. Region Based Binary Analysis

Prior to decompiling a function for patching, IRENE performs an analysis of each region to determine constraints on entry to, and exit from the region. These assumptions serve as the basis for cross region optimizations. These analyses are implemented in IRENE's binary analysis framework built on Ghidra [6]. A region's context contains two types of analysis information: live variables at entry and exit of the region, and constant values of variables at entry to and exit from a region. Additionally, the region analysis performs a type analysis to compute cross region type hints for the decompiler. This combination of information captures assumptions the decompiler needs in order to produce high-level liftings: dead variables that can be eliminated, variables' stack offsets in order to lift stack accesses to high-level variable accesses, and type information for rewriting pointer arithmetic [6].

These analyses are performed on a generalized fixpoint interface for intra-procedural control-flow-graphs (CFGs), enabling abstract interpretation based static analysis [7]. An analysis is then described as a set of transfer functions

over an abstract domain and P-Code operation [6]. IRENE uses two different domains: a liveness domain for live byte analysis, and a limited affine relations domain for a combined stack and value analysis. The liveness analysis is a traditional reverse dataflow analysis. Performing a live byte analysis allows IRENE to capture liveness in terms of sub-registers and consequently allows for more precise variable analysis.



ARAValue

$I = <\mathbb{Z}\times\mathbb{Z}, \text{hull}, \subseteq>$
T_m = Ideal integer domain transformer
$I_m = T_m(I)$
$D = \text{Varnode} \rightarrow \text{ARAValue}$

Fig. 3. Stack analysis domain

The stack analysis domain is based on the stack domain presented by Saxena et al. [8]. This domain relates registers to values that are either some integer value or an initial register value plus some integer offset. We make the modifications to the domain highlighted in Figure 3. The domain D maps Ghidra Varnodes that represent registers to ARAValues. In order to preserve soundness in the presence of machine integers with overflow and multiple signedness interpretations, IRENE uses an ideal integer transformer to transform an interval domain over ideal integers into an integer domain over machine integers of a width w. This transformation was originally proposed in Verasco, a C static analyzer, and allows for easy construction of transfer functions for analysis of machine integers without resorting to complex ad-hoc domains [9]. The domain transformer lazily restricts an integer value to the range valid for a given P-Code operation (e.g. restricting the value to the unsigned range $[0, 2^w - 1]$ for unsigned equality or division). This analysis collects both constant values, and values that are relative to the initial stack pointer or some incoming parameter.

The stack analysis results serve as an alias analysis during type constraint generation when recovering type information. Parameters with known types may be stored in a temporary location on the stack and then retrieved in a different block. The decompiler described in Section III-D cannot perform optimizations across basic blocks directly, while preserving sound patch localization. Instead, IRENE records this type information as a type assertion at the load instruction in the successor block. The type analysis and alias analysis derive these assertions. The type constraint generation is inspired by Retypd and walks the function, recording subtyping constraints between instructions [10]. The type solver is unification-based and reifies subtyping constraints to equalities with constraints for width subtyping for structure types in the style of HM(X) [11]. This simplification to unification is sufficient for the goal of propagating known types across blocks and does not result in propagating drastically over-refined types throughout the program due to the analysis being intra-procedural.

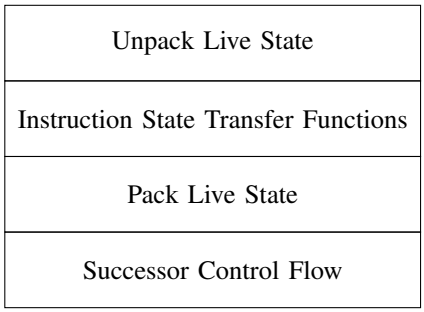


Fig. 4. The layout of a region function.

The type assertions, live variables, stack relations, and stack depth for each region are packaged into a region context. This region context, along with the instructions for the region, serves as the input to Anvill, IRENE's LLVM-based region decompiler.

D. Lifting Regions with Anvill

Anvill decompiles a region to LLVM IR in two main steps: lifting and optimization. LLVM IR provides a convenient substrate for recompilable decompilation, by ensuring that instruction semantics are recompilable through the LLVM compilation toolchain [5]. Anvill's lifting stage produces an LLVM function that represents a binary function by calling a sequence of region functions. A region function takes as input the function's in-scope variables, calls LLVM instruction functions corresponding to each instruction in a region, and then performs a region's exiting control flow. Exiting control flow can either call another region function or return a value and exit the function. The overall high-level function populates the variable state from the parameters of the function and then tail calls the entry region. This representation preserves the whole function while maintaining separate LLVM functions for each sub-function region. This separation is critical for ensuring that optimization does not break the localizability properties highlighted in Section III-B. By applying optimization independently to each region function, optimization only occurs internal to a region, using cross-region assumptions that are explicitly communicated to the optimizer through the region context.

A region function has four components, highlighted in Figure 4. The semantics of the block itself are represented as a sequence of LLVM calls to Remill instruction functions [12]. These functions consume the CPU's register state and memory, and modify the state according to the instruction's semantics. At the entry to a region function, live variables (passed as parameters to the region) are stored into the register and memory state. Affine relations and constant values are also stored into the state, expressing assumptions about the value of registers and memory upon entry to the region. Similarly, after the instructions' semantics have modified the CPU and memory state, the value of high-level variables is loaded from

the low-level state and stored into the variables that are live at exit of the region. By only storing variables that are live at exit of the region, the optimizer can eliminate computation that is not used by successor regions. Finally, the region performs the exiting control flow from the region. If the region can jump to another region, the region will call the successor region's representing function with the new live state.

The steps highlighted in Figure 4 produce a semantically valid representation of the binary region as a function over the live entry variables and live exit variables of the region. The representation at this point, however, is low-level, representing each instruction from the binary as a call to a function that implements that instruction. Anvill produces high-level decompilation by optimizing this representation in order to elide the CPU state, remove redundant computation, and simplify low-level idioms. The first step in optimization is to inline all instruction state functions. Then a combination of LLVM optimization passes and custom decompilation passes are run in sequence to produce high-level code. Important passes include LLVM scalar replacement of aggregates (SROA) which is responsible for decomposing the CPU state structure into single-static-assignment (SSA) values, the stack recovery pass which transforms loads and stores relative to the stack pointer (via the affine relations in the region context) into loads and stores of stack variables, the pointer lifter which rewrites pointer arithmetic to field accesses based on the region's type hints, and the branch recovery pass which rewrites comparisons based on CPU flags to high-level comparisons. The optimization infrastructure transforms low-level LLVM to high-level code.

E. PatchIR

The region LLVM function is not recompilable independent of the region context. The region context contains the definition of high-level variables in terms of low-level machine locations. This information is required to compile a modified region function to assembly that is compatible with the original binary. In order to provide a uniform representation of a region for a compiler, we developed PatchIR. PatchIR is a representation of region functions that contains both a region's semantics as an LLVM region function, and a region's context. The intermediate representation is built with the Multi-Level Intermediate Representation (MLIR) to enable the combination of LLVM operations with new custom operations [13].

Figure 5 highlights the structure of a PatchIR module. A PatchIR module reflects the structure of a binary through a collection of functions and regions localized to an address and size within the original file. Each PatchIR region describes the binary interface of the LLVM decompiled region function through `value` operations and a `call` operation. The call operation consumes a value defined by a value operation for each high-level variable that the LLVM region either consumes at entry to the region or modifies at exit from the region. A value operation holds the location of a variable at entry to and exit of a region, either in memory (as a value relative to the stack pointer) or in a register. A simplified example of a value operation where a high-level variable

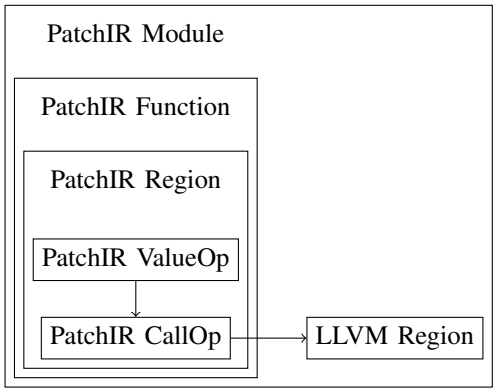


Fig. 5. A PatchIR Module.

local_2c is stored at a -44 offset from the stack pointer is: `value() {at_entry = [SP-44]: 32, at_exit = ["SP"-44] : 32>, name = "local_2c"} : () -> !ptr<i32>`. The value records the size of the value, its location, and its high-level type: `i32`. If the value is only used at entry to the region then the operation only contains the entry location, and if the value is only used at exit from the region only the exit location is set. The collection of value operations for a region call define the binary interface for that region in terms of the low-level locations for the function's parameters.

F. PatchLang

IRENE contains a domain specific language (DSL): PatchLang, for developing patches. Users can develop patches in an s-expression based DSL rather than directly writing LLVM. PatchLang has a direct correspondence with PatchIR, allowing lifting and lowering PatchIR modules to PatchLang. PatchLang provides convenience features for patch development like structured control flow (if, while), variables, and nested expressions.

G. Patch Compiler

The patch compiler consumes a PatchIR region along with the corresponding LLVM region and produces assembly compatible with that binary region's application binary interface (ABI) (defined by the region's value operations). The patch compiler executes a set of passes to derive an ABI compatible LLVM function, creates a custom calling convention from a region's value operations, and then invokes LLVM code generation for the given target. By keeping the process separate from code generation, the patch compiler can reuse LLVM codegen for any target that LLVM supports.

The key passes involved in producing an ABI compatible LLVM function are: parameter SSA transformation, type legalization, and relative reference replacement. To allow users to naturally express side-effects on variables, PatchIR regions consume pointers to these variables' locations that are stored to and loaded from. In order to convert these variables to ABI values, these pointer values must become direct, pass by-value parameters. The first pass in the patch compiler achieves this goal by wrapping each region in a function that

calls the region function with value parameters. These value parameters are stored into a structure and the callee region receives pointers into this structure. After LLVM SROA optimization, this wrapping function reflects the SSA version of the underlying region. After an SSA function is constructed, legal types must be assigned for each variable's parameter. A legal type is a type that can be stored in the low-level location associated with a parameter. The legal types for a given low-level location are defined by a TableGen-generated backend for each supported platform. Specifically, a patch compiler backend expresses the legal high-level and low-level types for a location, allowing conversion between the high-level type and legal type. Type legalization manipulates the signature of the region function to consume legal low-level types that are converted to high-level types prior to calling the original region function. Finally, the relative reference replacement pass replaces references to addresses in the binary with an address relative to a value holding the loaded base address of the binary. This reference rewriting pass allows the collaboration between the assembler described in Section III-H and the compiler in order to handle position independent code. The assembler populates the allocated base address value with the loaded base address of the binary so that references can be computed relative to this address. This pass also handles rewriting references to symbols in other libraries by using available global offset table (GOT) entries into the target library to compute the correct address for the target symbol.

H. Patch Assembler

The patch assembler is responsible for consuming the assembly representation of a recompiled region, the location of the region, and the original binary and producing a new binary with the region replaced with the new binary compatible code. The patch assembler achieves this replacement by introducing a detour at entry to the region to free space where the new region implementation will be placed. The restriction on regions as being single entry control-flow-free fragments in the original binary, means that by replacing the entry to the region, the region's functionality is replaced. The detouring functionality of the patch assembler is built with Patcherex2, a library for creating binary-level patches [14]. IRENE extends Patcherex2 with functionality to automatically create a patched binary from a patch definition generated by the patch compiler, providing the loaded base address to compiled code. Once the patch assembler has assembled the compiled patch and inserted the patch into the binary with a detour from the target location, the binary is patched and can be redeployed.

IV. Case Study

In order to evaluate the efficacy of IRENE patches, we performed a case study consisting of developing a micropatch for a vulnerability in an open source embedded operating system. Selecting a pre-existing vulnerability in open-source software allows us to compare the size of the IRENE micropatch to source-based patching scenarios where a different toolchain, different version, and same toolchain and version are used to patch a vulnerability. We also compare the IRENE decompilation patch to a handcrafted assembly patch.

A. The Vulnerability

We produced a patch for CVE-2023-24819 in RIOT-OS's generic network stack [15]. This vulnerability is a remote buffer overflow in the function `gnrc_sixlowpan_iphc_recv` that handles compressed IP headers in the 6LoWPAN protocol. When receiving a 6LoWPAN fragment, the function does check that the fragment size does not exceed the expected size for the reassembled packet (and therefore the reassembly buffer). This check, however, does not account for the size of the IP header after decompression. An attacker can therefore achieve an out-of-bounds write by sending a fragment that exceeds the size of the reassembly buffer by less than the size of the decompressed IP header. The upstream patch for this vulnerability adds an additional check when processing a fragment that checks the size of the fragment after header decompression.

B. Comparison

To compare the size of IRENE-based micropatching to source-level alternatives we compare IRENE to three scenarios: a scenario where an engineer has access to both the original toolchain and source code on a target device, a scenario where an engineer only has access to a different version of the toolchain, and a scenario where an engineer only has access to a newer version of the source code. To record the extent of changes between a patched and unpatched binary we record the sum of the Levenshtein edit distance between each function in the original and patched binary. We used RIOT-OS version 2022.07 for our unpatched baseline. We used ARM Embedded toolchain version 10.3.1 as our baseline toolchain version. We used version 2022.04 as our alternative RIOT-OS source code version and ARM Embedded toolchain version 9.2.1 as our alternative toolchain version.

C. Results

TABLE I
PATCHING SCENARIO RESULTS

#	Compiler	Source Version	Method	Distance
1	10.3.1	2022.07	IRENE	61
2	10.3.1	2022.07	Recompilation	275
3	10.3.1	2022.04/07	Recompilation	1323
4	10.3.1/9.2.1	2022.07	Recompilation	6767
5	10.3.1	2022.07	Manual	20

Table I highlights five patching scenarios. For scenarios where the version is varied between the baseline and patched version, the versions are displayed as *baseline version/patched version*. The results in Scenario 1 highlight IRENE's capability to affect the patch on an arbitrary version of the program without access to the same source code or toolchain with a distance of only 61 bytes from the original program. This distance is smaller than even the changes introduced by Scenario 2 which introduces the patch on the same version

of the program with the same toolchain version. The size of the patch in Scenario 2 is caused by the patch resulting in different code generation within the patched function: `gnrc_sixlowpan_iphc_recv` (51 changes) and subsequent changes to jump addresses to compensate for the increased size of the function (224 changes). Detour-based patching avoids the need to repair jumps by maintaining the location of all jump targets within the original binary. Scenario 3 demonstrates the advantage of IRENE patching when only a newer version of source code is available than the one currently running on the target device. Comparing the recompiled patch to the most recent prior version of RIOT-OS resulted in a distance of 1323 and 13 unmatched functions. Similarly, scenario 2 demonstrates the high degree of variability introduced by changing to a different toolchain version when the original toolchain used to build the program running on the target device is not available. By patching the binary directly, IRENE can produce a small patch directly on a target binary.

Finally, we compare IRENE to Scenario 5 where we introduced a patch written manually in assembly. The size of this patch is smaller and can be applied to the binary directly, using Patcherex to introduce a detour. This process, however, is manual and requires expertise in reverse engineering.

```
ldr.w r9, [r6, 8]
adds r3, r4, r2
cmp       r3, r9
bhi       26452
```

Fig. 6. Assembly Micropatch for CVE-2023-24819.

Figure 6 shows the assembly patch for the vulnerability. For an engineer to successfully patch the vulnerability, the user must identify that `r9` is a free register available for use, `r6` contains the target IPv6 buffer and the offset to the size field is 8 bytes, `r4` contains the uncompressed header size, `r2` contains the size of the incoming packet payload, and know the ARM thumb assembly required to implement the semantics of the C patch.

```
(let pkt_size (load type: i32
   (getelementptr
      (load type: ptr named_val_549_ipv6)
      gnrc_pktsnip  0:32 2:32)))
(let total_write_size
   (add (load type: i32 uncomp_header_size)
   (load type: i32 packet_size)))
(cond_goto 26452:32
   (ugt total_write_size pkt_size))
```

Fig. 7. IRENE patch for CVE-2023-24819.

Figure 7 shows that IRENE allows an engineer to operate over a higher level of abstraction when writing a patch. As long as variables are recovered within Ghidra, IRENE allows the user to access those variables (e.g. `packet_size`). The user can use the layout of `gnrc_pktsnip` to access the size field of the IPv6 buffer.

V. Conclusion and Future Work

Patching embedded devices with long life-cycles in the field requires new techniques to produce patches that are small and consequently easier to verify. IRENE's approach of sub-function decompilation allows engineers to develop small targetted micropatches that leverage detouring at a higher level of abstraction than previously possible. Further optimization of the approach, including a non-position-independent mode, could allow for smaller and easier to verify patches that do not use indirect control flow.

References

[1] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware." in *NDSS*, vol. 1, 2015, pp. 1–1.

[2] G. J. Duck, X. Gao, and A. Roychoudhury, "Binary rewriting without control flow recovery," in *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*, 2020, pp. 151–163.

[3] 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, pp. 1497–1511.

[4] 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 *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 133–147. [Online]. Available: https://doi.org/10.1145/3373376.3378470

[5] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[6] "NationalSecurityAgency/ghidra: Ghidra software reverse engineering (SRE) framework." [Online]. Available: https://github.com/NationalSecurityAgency/ghidra

[7] P. Cousot and R. Cousot, "Abstract interpretation frameworks," *Journal of logic and computation*, vol. 2, no. 4, pp. 511–547, 1992.

[8] P. Saxena, R. Sekar, and V. Puranik, "Efficient fine-grained binary instrumentationwith applications to taint-tracking," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, 2008, pp. 74–83.

[9] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie, "A formally-verified c static analyzer," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 247–259. [Online]. Available: https://doi.org/10.1145/2676726.2676966

[10] M. Noonan, A. Loginov, and D. Cok, "Polymorphic type inference for machine code," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 27–41. [Online]. Available: https://doi.org/10.1145/2908080.2908119

[11] M. Odersky, M. Sulzmann, and M. Wehr, "Type inference with constrained types," *Theory and practice of object systems*, vol. 5, no. 1, pp. 35–55, 1999.

[12] "lifting-bits/remill," Apr. 2024, original-date: 2015-10-22T13:54:19Z. [Online]. Available: https://github.com/lifting-bits/remill

[13] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 2–14.

[14] "purseclab/Patcherex2," Apr. 2024, original-date: 2023-11-22T20:30:12Z. [Online]. Available: https://github.com/purseclab/Patcherex2

[15] "Buffer Overflow during IPHC receive." [Online]. Available: https://github.com/RIOT-OS/RIOT/security/advisories/GHSA-fv97-2448-gcf6