EMULATION BASED SECURITY MEASUREMENT WITH APPLICATIONS IN AVIONICS, REDACTIONS, AND INDUSTRIAL CONTROL

BY

MAXWELL BLAND

B. S., University of Califonia, San Diego, 2018
M. S., University of Califonia, San Diego, 2019

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2023

Urbana, Illinois

Doctoral Committee:

Professor Kirill Levchenko, Chair and Director of Research
Professor Adam Bates
Professor Aaron Schulman
Professor Gang Wang

Abstract

The saftey of critical systems and data is of paramount importance to society. Attacks on these systems can have catastrophic consequences, and the security of these systems is often difficult to measure. Existing methods often involve access to ground-truth information, which is not always available. In this dissertation, we explore the use of emulation to measure the security of systems in the absence of ground-truth information.

Complex system emulations often require sophisticated approaches to digital forensics and tactics for navigating undecidability resulting from uncertainty of the system's state. To address these challenges, we present intelligent guess and check strategies for deducing hidden information in symbol-stripped binary firmware. Our core technical contributions are (1) the first abstract interpretation based firmware rehosting system, used to generate emulations of embedded systems. (2) A novel system for the analysis and recovery of glyph positioning information in PDF documents. This system was used to recover redacted text information where the characters were removed in hundreds of sensitive documents. (3) A logic-based intermediate representation and framework for the extraction of lifted function summaries from binary firmware. This framework makes existing verification and synthesis techniques applicable to real-world systems by translating implemented code to mathematical models. Where appropriate, we justify our strategies through discussions of correctness, precision, and generalizability. Our results are never theoretical: we apply them to pre-existing, empirically validatable domain rather than models: among others, we study the Communication Management Unit used in Boeing 737 Aircraft, historically important redacted documents, and a programmable logic controller operating a Tennessee Eastman chemical plant reactor pressure valve.

Acknowledgments

I would like to thank my advisor, Kirill Levchenko, for pushing me to standards and achievement few attain and for maintaining his support throughout the Ph.D. process. Thank you to my committee members, Adam Bates, Aaron Schulman, and Gang Wang, for their support and guidance throughout the Ph.D., before, and in the completion of this disseration.

Thank you to Abraham Clements, Gabriela Ciocarlie, and Richard Kennell with CyManII for their support, advice, feedback, particularly in relation to the InteGreat system. Additionally, I thank my collaborators Anthea Chen, Anushya Iyer, Stephen Checkoway, Stefan Savage, Joshua Mason, YiFei Zhu, Evan Johnson, Mingjia Huo, Stewart Grant, Alex Snoeren, Anil Yelam, and Nishant Bhaskar for their support and efforts throughout our papers together, though the subjects are not all approached in this thesis. I also thank the University of Illinois, National Science Foundation, and CyManII for funding my research and efforts as a graduate student.

Additional thanks are in order for my labmates Tzu-Bin Yan, Margie Ruffin, Michael Chen, Megan Culler, Jonatas Marques, and many others at the Coordinated Science Laboratories who have provided support and friendship throughout my time at the University of Illinois, including Andrew Miller, Nikita Borisov, Michael Bailey, Deepak Kumar, Zane Ma, Paul Murley, Kyo Kim, and Joshua Reynolds. I would also like to thank my friends in the Urbana-Champaign community, including Kevin Langowski, Alex Rogers and the Ship of Fools, the Rose Bowl, Waluigi's Mansion, the Spice Rack, and other basements unnamed who supported my harsh noise wall and other musical experiments.

Finally, thank you to Annika Sornson, whose love made me realize not all systems need to be broken: unfortunately, it was too late for the redactions, airplanes, and others. Thank you as well to her parents for their support and friendship. Finally, thank you to my parents and step-parents for their love and sacrifices. Everyone mentioned above is impossible to emulate.

"The revolutionary character of scientific knowledge is its speculative import."
— Quentin Meillassoux, *After Finitude, p. 120.*

"Reason is, and ought only to be the slave of the passions."
— David Hume, *A Treatise of Human Nature, II.3.3, p. 415.*

"We only have a right to the retro, to the phantom, parodic rehabilitation of all lost referentials."
— Jean Baudrillard, *Simulacra and Simulation, p. 41.*

Table of Contents

Chapter 1: Introduction

Software systems, usually invisible to the end user, are ubiquitous in our daily lives. They are used in a wide range of applications, from consumer electronics to critical infrastructure. The security of these systems is paramount, as they are prevalent in our daily lives, however, analyzing these systems can be difficult as code and hardware may be opaque, meaning vulnerabilities may be known to specialized attackers that are not possible to measure automatically or easily. Existing techniques often make strong assumptions about our understanding of the system, such as access to source code, non-symbol-stripped binaries, or mathematical models.

Conventional approaches rely on real-world testing or complex simulations, bug reports, high-level verification (formal methods), and manual analysis to find bugs and vulnerabilities. There is nothing inherently wrong with any of these approaches, but they still have limitations due to complexity, cost, coverage, and effort involved, limiting these approaches to parties with sufficient resources to perform them. And regardless of the approach taken to security, as with most problems, there remain unknown features of the system under analysis (or simply too many systems), making it potentially impossible to predict all future exploits. In 2022, Mitre reported 34,553 CVE entries [1], following a steady, monotonic increase from 2016. Moreover, metrics for evaluating the complexity of source code are still fail to capture developer expectations [2, 3, 4]. Even in the "good" case, where we have perfect ground truth for a system, algorithms struggle to make accurate judgements of general constructs in real systems, judgments necessary to determine systems' security.

The security of a system hinges on whether the flaws in the system are represented in an accessible medium whereby they are effectively identified and corrected, a hard problem as every measurement exists in a specific interpretive context [5]. Each paradigm for securing a system, e.g. a provably correct decision system, implicitly determines which flaws are considered or prevented and which are overlooked. Ultimately a variety of approaches are necessary to secure a system and all of them, in principle, make some statement about the observable behavior of the system during operation. Security is a veritable symbolic arms race towards better representations and emulations, as evidenced by recent works [6, 7, 8], including works from this dissertation [9, 10, 11]. In this dissertation, we present several novel approaches to the representation and emulation of complex, opaque software systems, broadly applicable and empirically capable of identifying flaws in and exploits for real-world avionic, PDF document redaction, and

industrial control systems.

Terminology This dissertation focuses on single, specific concept of *emulation* as applied to systems. We define emulation as the process of executing a model of the system where at least one component is not represented during simulation on some kind of host computer [12]. Perfect *simulation*, on the other hand, involves modeling the entire underlying state of the target. Implied in this definition is the fact that emulation embraces speculation, non-exact representation, and a lack of ground truth for the system being analyzed, making it more broadly applicable than simulation for complex systems. Emulations, simulations, and the systems in question are subjects of *dynamic analysis*, which attempts to run the program and analyze its execution, introduced by Ball in 1999 [13]. Dynamic analysis is contrasted with *static analysis*, which attempts to analyze the program without running it and also attempts to make statements about the dynamic behavior of the system [14]. In either case, the purpose of these analyses is typically to enforce some requirement (also called property or specification), to represent the desired behaviors of the system, such as safety, stability, or temporal constraints, falling into the subdomain of formal methods [15, 16]. More generally though, emulations can be used to imperfectly replace the system in question, aiding in processes falling outside the purview of verification, such as binary exploit development, information leak detection, and behavioral prediction [17, 18, 19].

Within the broad areas of emulation and simulation, this dissertation focuses on three strategies or techniques: *rehosting*, *dictionary attacks*, and *lifting*.

Rehosting Rehosting is the process of bootstrapping a system so that it can be emulated in a different context, typically a laptop computer instead of the original embedded system, e.g. a programmable logic controller. This encounters the particular challenge of modeling or reimplementing subsystems or subcomponents that are not present in the new context, such as hardware devices. If a ground truth reference for the subsystem's behavior is available, then this can be used to bootstrap a simulation of it (or sometimes a connection to the subsystem itself) in the new context, but this is not always the case. In cases where a ground truth reference is not available, the behavior of the subsystem must be inferred from residual information recoverable from the system, such as binary code for interacting with the subsystem.

Dictionary Attacks A brute force approach to emulation that attempts to perform a universal quantification of the possible true behaviors of the system and then use some

checking or verification mechanism to isolate the correct behavior. In many cases it is possible to extract a function from a program, and evaluate it in isolation of the rest of the system for all possible given inputs, recovering all possible outputs. We also classify fuzzing, the process of mutating inputs to a system to find unexpected behaviors, as a form of dictionary attack. This can precisely determine what the system is capable of in circumstances where a specific function can be isolated, the number of possible inputs is constrained, and a precise verification mechanism is available.

Lifting Lifting is the process of translating the representation of a system into a different symbolic domain which is easier to reason about. The result of lifting is an intermediate representation or extracted model of the system. An example of this is a translation from binary code to mathematical statements which can be emulated outside of the complex context of software, e.g. one with hardware device models. While this can potentially miss the validation of critical components, careful lifting can also avoid the necessity of reasoning about unknowns in the system.

Challenges Rehosting, dictionary attacks, and lifting each have their own challenges in generalization, scalability, and completeness respectively [20, 21, 22]. Significant progress continues to be made in addressing these challenges in order to apply each of these techniques to practical problems, and approaches to emulation are often mixed in order to acheive effective results [23, 24, 25, 26, 27, 28]. However, each of these strategies presents a perspective on the problem of evaluating a system in the absence of a complete model, and so all recent work shares a common goal of determining knowable and unknowable features or flaws in a system.

Dissertation Topics Whereas verification enforces a specific representation of a system through mathematical proof, emulation strives to develop a useful clone of the system for another context. More recent data-driven verification approaches bridge this gap but do not approach the subject of emulation and how it affects modeling. This dissertation does not provide a framework for completing all incomplete system models but does make progress the above common goal by introducing theories on or strategies for the emulation of complex systems. One shared theme is the use of subtle information present in things which are distinctly not the system in order to emulate the system. As a result, our strategies are particularly useful for real-world systems of which only some qualities are known or accessible. Thus, we are also able to provide evidence of our theories' effectiveness in practice and provide insight into the absolute limits of emulation.

In the description of these strategies for emulation, we will find the application of guesswork is unavoidable, and that these guesses must later be checked and verified against the system itself. Correct emulation avoids making unjustified assumptions about possible states or accurate respresentations of the system under analysis. Luck and probability play an integral role in the application of emulations and the proper use of emulation often requires the joint application of uncertainty and sensitivity analysis, but we will not limit ourselves to analyzing emulations in terms of axiomatics or quantification via a unifying theory. A complete description of the problem acknowledges mathematics does not confer existence upon its subject and adopts conceptual or empirical strategies into the execution and analysis of unknown system features. That is, in this dissertation, we focus on the symbolic representations that exist prior to analysis and effects of interpretive frameworks on the generation of emulations from the context of:

1. the synthesis of hardware models from software information [9] (Chapter 4),
2. the determination of text content after it is removed (redacted) from PDFs [10] (Chapter 5), and,
3. the use of logic-based intermediate representations for lifting between symbolic domains [11] (Chapter 6).

With exceptions for the implementation of a taint-tracking based symbolic execution system in QEMU and some ideas relating to symbolic search strategies (Chapter 4), the information leak quantification for specific redaction dictionaries (Chapter 5), as well as the analysis of the quadcopter stabilization algorithm (Chapter 6), which were implemented jointly with coauthors, the works presented in this dissertation are the author's own with dialectic support from coauthors.

We next provide a more detailed introduction to each of the above topic areas, followed by a summary of contributions and impacts.

1.1 TARGETED FIRMWARE REHOSTING FOR EMBEDDED SYSTEMS

The possession of a physical copy of a critical system for analysis is often difficult. Additionally, it is valuable to attain an emulated copy of these systems for the purposes of precise, invasive analysis without binary rewriting. Implementing an emulation of a system's hardware devices is labor intensive, therefore, automated methods for generating emulated, rehosted copies of systems are valuable.

In this dissertation, we discuss the possibility of generating emulations of hardware from information present in firmware alone, with the first strategy being the symbolic analysis of firmware interactions with hardware devices. By collecting the constraints software places on the function of hardware during interaction with hardware peripherals, it is possible to synthesize a model of the hardware device. We show the limitations and challenges involved in this analysis and find that in many cases it is possible to use this partial constraint information to generate a useful emulation of associated hardware. In solving this challnge, we introduce one of the first fuzzing systems targeted at arbitrary QEMU system simulations, and several tactics for dealing with undecidable problems related to symbol-stripped, binary program analysis. We applied this system to more than ten firmware images, but we focus on one in particular, the Communication Management Unit of a Boeing 737, and other aircraft, responsible for managing a large part of the communication of key Line Replacable Units (LRUs) in the aircraft, including the Flight Management Computer. Our emulated hardware not only allowed us to develop an effective rootkit and local code execution attack on the CMU, but also identify three remote messages capable of powering down the machine, creating a denial-of-service attack.[1]

In the preliminary section of this dissertation, we introduce the notion of abstract interpretation as a method of reasoning about symbols inside of algorithms. This method was implied in the form of symbolic execution within the Jetset system [9], where it was able to rehost all the devices targeted by an "equivalent" fuzzing approach [23], as well as and SEL Feeder Protection relay controlling breaker switches in electrical substations [29], the aforementioned CMU-900 [30], and various SoCs (Raspberry Pi 2 and BeagleBoard-xM) [31, 32]. The rehosted emulations can be subject to several forms of potentially destructive dynamic analysis routines that would be impossible on the physical device, such as fuzzing, which may "brick" the device in question. The use of symbolic execution for rehosting-related tasks has now been adopted and integrated into several additional research works in top venues to rehost a variety of systems [33, 8, 34, 35]. A complete explication of the challenges and techniques involved in rehosting is beyond the scope of this disseration and we refer interested readers to [20].

[1] Due to their sensitivity, the specifics of these exploits are not given explicit detail.

1.2 GLYPH POSITIONS BREAK PDF TEXT REDACTION

The analyis of embedded systems is just one application of emulation—emulation of software systems can also play a vital role in the analysis of information leaks. By extracting and emulating the glyph positioning behavior of popular PDF document creation software, it is in fact possible to generate precise fingerprints for the layout of text on a page. It turns out that this layout contains sub-pixel error-correction information which leaks a large amount of information about any text redactions where characters are removed the document's digital representation.

To emulate these different PDF creation software stacks and break redactions, we developed a novel framework Edact-Ray [10] which attempts universal quantification over potential redacted texts in emulation to identify the input which produced the PDF document under analysis. The system also includes scripts which are capable of analyzing thousands of PDF documents and locating vulnerable redactions in these documents in bulk. Using access to the RECAP court document system [36], we were also able to identify hundreds of non-excising redactions where the text was simply covered with a black box rather than being removed from the PDF, resulting in the discovery of confidential trade secrets for large corporations, the addresses of famous US celebrities, and salacious statements. This has resulted in the notification of hundreds of lawyers involved and significant efforts to make redaction uniform across court systems.

We used the Edact-Ray system to break a number of historically relevant redactions otherwise thought to be secure. These include redactions in documents from the Digital National Security Archives (DNSA) [37], and the US Court System [38]. We also used the system to identify hundreds of broken redactions in Office of Inspector General (OIG) [39] and Freedom of Information Act (FOIA) documents [40]. Our findings resulted in the notification of more than 22 different US government agencies, national media recognition [41] and extensive collaborations on working towards a solution to the problem of broken redactions.

1.3 LIFTING CONTINUOUS CONTROL EQUATIONS FROM BINARY CODE

The last topic addressed in this dissertation is a bottom-up approach to verification, a system for lifting discrete binary code algorithms to the continuous domain for simulation in frameworks like Matlab Simulink [11, 42]. We consider the possibility of representing symbolic translation rules in a logic-based intermediate representation, where microarchitectural features are modeled deductively and complex statements about pro-

gram semantics may be derived by nesting simpler statements. Intermediate representations for the purposes of lifting (decompilation), translation, and emulation are an active area of research and are used by several high-powered offensive analysis tools such as angr and Ghidra, however, logic has not yet been used as a syntax for describing microarchitectural features in a decompiler [43, 44, 8, 45, 46, 47].

As a result, current intermediate representations for binary analysis suffer from several drawbacks related to nesting and translation. They strive for a static representation of program semantics rather than one which must be dynamically evaluated, limiting their representational capacity for more complex semantic structures present in code. This dissertation explores the possibility of using a logic-based intermediate representation to lift continuous control equations from the binary firmware of Cyber-Physical Systems [48]. We begin by representing and lifting simple microarchtectural semantics, such as the splitting of a floating point number split across registers 0 and 1 in the ARM architecture. These rules are then nested into more complex deductive chains, such as the structure of common mathematical functions, through the use of statements in propositional logic over the theory of uninterpreted functions[2]. The representation is extracted using abstract interpretation as a subroutine to identify the specific relations between input and output locations, in order to translate logically-defined program slice boundaries in the system under analysis to relations and operations over uninterpreted functions. The results are then combined via a theorem prover to lift implicit, abstract semantics from a binary firmware images, e.g. mathematical functions like cosine.

We implemented this approach in a tool called InteGreat and use the tool to extract functions from a Programmable Logic Controller (PLC) used to control the reactor pressure of a chemical plant, as well as the stabilization algorithm of a quad-copter and a single backpropagation step in an artificial neural network. InteGreat does not just lift program slices to the continuous domain using the logical statements embedded in its intermediate representation: it then translates the lifted representation into a Matlab emulation of the system, allowing us to model precise destabilization attacks on the chemical plant and identify flaws in implementations and published versions of the quad-copter's stabilization algorithm.

[2]Uninterpreted functions discard the computation under analysis and reduce it to a single symbol and list of input, output locations.

1.4 SUMMARY OF CONTRIBUTIONS

The effective emulation of complex systems is a critical component of their analysis and measurement. This dissertation presents three different approaches to the emulation of complex systems, each of which has been used to identify and exploit vulnerabilities in real-world systems, and its objective is to expand our understanding of how unknown information in systems may be rediscovered. Our contributions are therefore as follows:

1. Novel state-of-the-art techniques in the rehosting and analysis of complex embedded systems through the use of targeted abstract interpretation and comprehension of firmware constraint sets into models of hardware peripherals. This approach allowed us to emulate hardware devices across three different miroarchitectures [9], and perform dynamic analysis (fuzzing) of the systems in question, then reproduce the results of fuzzing on the related physical devices. We were able to develop a sophisticated exploits for a critical avionics system by combining careful analysis of the generated emulations with analysis of the device in question [49] (Chapter 4).

2. The first effective attacks on document redactions where the characters of the text are not presentin the PDF document (a literal black box) [10]. By emulating the glyph positioning behavior of popular PDF document creation software, we were able to uncover sensitive information from several documents of significant historical importance. Our approach has been used to identify hundreds of broken redactions in documents from the US Court System, the Digital National Security Archives, and the Office of Inspector General (Chapter 5).

3. The introduction of logic-based intermediate representations for binary lifting and the bottom-up verification of cyber-physical system implementations [11]. We implement a system adopting this approach and use it to extract control equations from real-world firmware, emulate the associated systems behavior, and develop a precise destabilization attack on a chemical plant's reactor pressure (Chapter 6).

To the extent that it is safe, we have released software code and tools for all associated techniques: JetSet [50], which applies symbolic execution to rehosting, Edact-Ray [51], but only the portions relating to the location and defense of vulnerable redactions, and InteGreat [11], for lifting using a logic-based intermediate representation. Each of these systems has been applied to real-world case study systems with verifiable results. The most impactful of these is Edact-Ray, which has identified vulnerabilities and hidden information in a number of important documents (Sec. 5.4 and Sec. 5.5); the full version

of the tool now sees continued use by the US courts and the US OIG. Jetset was able to sucessfully rehost thirteen distinct peices of firmware (Sec. 4.2) and an associated system-mode QEMU fuzzer was able to reproduce and identify identical codepaths across real and Jetset-emulated copies of both Linux system calls on the Raspberry Pi 2 and VRTX system calls on the CMU-900 (Sec. 4.3.2). The techniques and tools are also actively being shared with numerous other researchers in the Department of Energy, Universities, and private industry.

1.5 DISSERTATION STRUCTURE

We introduce technical concepts and background in Chapter 2. Chapter 3 describes different forms of emulation and the systems used throughout this dissertation, such as symbolic executors, as well as related work on each subject. We then introduce the concept of rehosting for emulation in Chapter 4, and describe the JetSet system for rehosting embedded systems. Chapter 5 describes emulation's role in dictionary attacks on leaked information as well as the Edact-Ray system for identifying broken redactions in PDF documents. In Chapter 6, we introduce the InteGreat system for lifting control equations from binary firmware and the function extraction, summarization for the purposes of emulating of complex cyber-physical system behavior. We conclude in Chapter 7 with a summary of this dissertation and a discussion of future work.

Chapter 2: Preliminaries

Here, we introduce concepts, vocabulary terms, operations, and initial related work that is referenced throughout this disseration. The theme of each of these subject areas is the representation of information, whether that information relates to the operation of binary code, the content of removed text, or a deduced interpretation of an existing system.

2.1 ABSTRACT INTERPRETATION

Under the traditional definition, *abstract interpretation* is a method of approximating the behavior of a program using monotonic functions over ordered sets (typically lattices). It is related to the emulation of a program in that it is intended to extract a model of the program's behavior without performing the exact computations involved. However, abstract interpretation has become more of a general framework for defining relations between different domains, as the necessity of translation between dissimilar symbolic representations, e.g. in transpilers, has become more prevalent.

Arbitrarily, let L and L' be *concrete* and *abstract* ordered sets, repectively.[1] These sets are related to each-other by a total function, which maps elements in one set to another. We define two functions, α and γ, which map elements of L to L' and vice versa, and are called the *abstraction* function and *concretization* function respectively.

Then, given ordered sets L_1, L_2, L'_1, L'_2, the concrete semantics f is a monotonic function from L_1 to L_2, and a correct abstraction from L to L' is *valid* if the abstract semantics f' mapping L'_1 to L'_2 preserves the ordering in f of L_1 to L_2. In this definition the ordering operation on the sets is left undefined, and in the absence of a decidable set of mathematical qualities, e.g. many nontrivial computer programs, so standard implementations of abstract interpretation on program execution operate by mapping between a concrete execution trace in registers and memory and the theory of bitvectors, rather than the highly abstract semantics used by programs' users and authors.

[1] The ordering of higher or lower levels of abstraction themselves is a function over the mathematical structures being discussed.

2.1.1 Symbolic Execution

The specific form of abstract interpretation from machine code to the theory of bitvectors is often referred to as *symbolic execution*, and is a method of program analysis that operates by tracking the values of variables in a program as symbolic expressions, rather than concrete values. For example, in standard execution the statement $(x > 5)?\ y + 1 : x + 1$ would be concretely evaluated to 4 if $x = 3$, however, a standard symbolic execution, the system will record the pair $(x <= 5; x = x + 1), (x > 5; x = y + 1)$ into a symbolic state, representing the two possible execution paths.

Symbolic execution is a powerful tool for program analysis, but it is not without its limitations. The core challenge is the *path explosion* problem, where the number of possible execution paths grows exponentially with the number of branches in the program. This can be limited through *loop invariant* analysis, where the system attempts to identify the invariants of a loop and only execute the loop a finite number of times, a *search strategy*, which determines which execution paths to explore from the exponential number possible, or a number of other deductive techniques. Closely related is the challenge of *memory aliasing*, which occurs when the system is unable to determine whether two pointers point to the same memory location, as the values of each pointer are treated as symbolic expressions over the theory of bitvectors rather than concrete values. Additionally, there is the *environmental modeling* problem, where symbolic execution is unsound due to missing context, typically because the specific execution behavior of the hardware is not present, e.g. interrupts or highly-specific microarchitectural features.

2.1.2 Lifting

We contrast abstract interpretation with the notion of *lifting*, a borrowed term concept from category theory, where given a structure-preserving map from one mathematical structure to another of the same type (a morphism), $f : X \to Y$, and another $g : Z \to Y$, a lifting of f to Z is a morphism $h : X \to Z$ such that $f = g \circ h$, or more simply, a function that points to another representation that *may* more closely resemble the structure Y but also has a morphism to Y (Fig. 2.1).

When applied to code, lifting typically refers to exposing the semantic structure of the binary to a higher level of abstraction. However, as the mathematical definition suggests, lifting may also refer to a *different* structure with the same morphism to the "true" structure. Note that lifting is a form of *translation*, but the new structure exists over a different domain. For example, machine code may be lifted to a symbolic representation

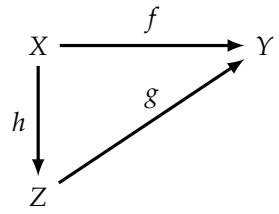


Figure 2.1: Lifting f to Z

that represents a call-graph, or it may be lifted to a higher-level language, e.g. C, but it would be translation rather than lifting if the code was made into an equivalent, different machine code, e.g. ARM to x86.

Standard forms of lifting include translating a program into a form of graph representing latent relationships in the program, in order to simplify algorithms for reasoning about the program's structure. For example, a program may be translated into a *control flow graph* (CFG), where each node represents a basic block of code and each edge represents a possible transition between blocks. Figure 2.2 shows an example of lifting a CFG from a simple machine code program.

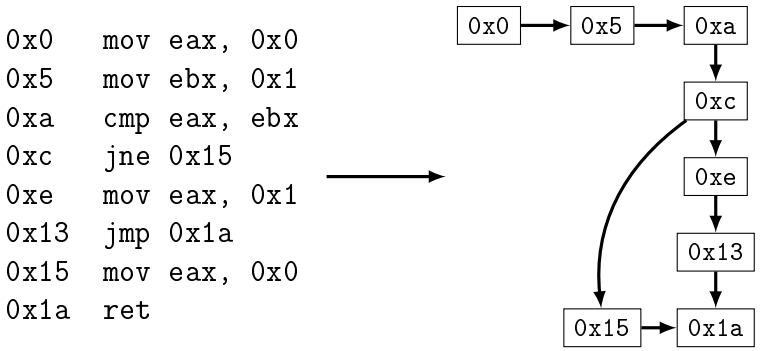


Figure 2.2: Lifting a CFG from machine code

Program slicing is a form of lifting which transforms a program specification into *slices*, sets of statements affecting sets of values in the program's state at some points of interest. More generally, they may also refer to arbitrary subsets of the program's statements. The determination of the slices is typically referred to as a *slice criterion*. Symbolic execution can be used to determine a given slice by exploring all paths through a program and thus the statements which affect the value of a variable of interest.

2.1.3 Intermediate Representations

The term *intermediate representation* (IR) refers to a representation of a program that is used as an intermediate step a translation process, typically for a compiler. The IR is typically designed to be easy to translate to and from the source and target languages, and to be easy to reason about. Referring to Fig. 2.1, in lifting the IR is the structure Z used to translate between X and Y. The critical form of IR for use in symbolic execution and abstract interpretation more generally are abstract syntax trees (ASTs). ASTs are a form of IR that represent the syntactic structure of a program as a tree, where each node represents a syntactic construct and each edge represents a relationship between constructs. For example, the AST for the expression `a + b * c` is shown in Fig. 2.3.

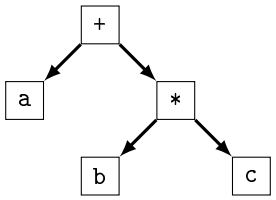


Figure 2.3: AST for `a + b * c`

ASTs may be used in conjunction with *Hoare logic* to reason about programs. Hoare logic is a formal system for reasoning about the correctness of programs. It is based on the notion of *Hoare triples*, which are statements of the form $\{P\}S\{Q\}$, where P and Q are *predicates* and S is a statement. The meaning of a Hoare triple is that if the predicate P holds before the execution of S, then the predicate Q will hold after the execution of S. For example, the Hoare triple $\{x = 0\}x := x + 1\{x = 1\}$ states that if x is equal to 0 before the execution of $x := x + 1$, then x will be equal to 1 after the execution of $x := x + 1$.

This is equivalent to a *satisfaction relation* between a program and a specification, which evaluates to true if the program satisfies the specification. In symbolic execution, this specification is represented as a set of predicates defined over the program's state, determined both by external inputs and the current execution path under consideration. By evaluating the satisfaction relation, symbolic execution can determine whether a given path is feasible.

Finally, note that while symbolic execution, lifting, and intermediate representations *can* preserve a program's semantics, each has the potential to introduce errors. For example, the process of lifting a program to an IR may introduce errors if the IR is not expressive enough to represent the semantics of the program. The outputs of this translation ultimately serve to perform analysis on the program's behavior, and the correctness

of this analysis depends on the correctness of the translation. Thus, the use of these techniques is ultimately one basis for emulation, an insight that will become more apparent in our discussion of QEMU (Sec. 3.1).

2.2 INFORMATION

Many definitions of *information* exist—this dissertation will use information to represent *pure difference* or distiguishability between two objects, e.g. 0 and 1. A key measure of information is *entropy* which quantifies the amount of uncertainty involved in a random process. Given a random variable X with a probability distribution P, the entropy of X is defined as

$$H(X) = -\sum_{x \in X} P(x) \log_2 P(x) \tag{2.1}$$

where $P(x)$ is the probability that X takes on the value x. For example, the entropy of a fair coin is $H(X) = -\frac{1}{2}\log_2 \frac{1}{2} - \frac{1}{2}\log_2 \frac{1}{2} = 1$. The entropy of a biased coin with $P(\text{heads}) = 0.9$ is $H(X) = -0.9 \log_2 0.9 - 0.1 \log_2 0.1 \approx 0.47$. The use of base 2 for the log indicates that entropy is measured in *bits*. Entropy quantifies the amount of information in a distribution as the number of bits required to represent it. For a uniform distribution over n values, the entropy is $\log_2 n$.

It is therefore also possible to quantify the amount of information *leaked* by a process if we consider the amount of information present in the initial, presumed inaccessible distribution and compare this with the amount of information present in the final, accessible distribution. What actually exists is what can be differentiated: if the number of bits in the final distribution is lower than the number of bits in the initial distribution, then some information has been lost. This does not necessarily indicate it is impossible to disambiguate a *specific* value using accessible information, but (assuming quantization) that some value from the initial distribution cannot be differentiated from some other value.

We will refer to the prior, inaccessbile distribution as a *dictionary* which limits the scope of the possibilities we are willing to consider. In some cases it is possible to construct a dictionary that is *complete*, i.e. that contains all possible values. When a dictionary is incomplete, often due to practical limitations on computation and analysis time, it becomes necessary judge whether a dictionary is reasonable.

2.2.1 Guess and Check

We consider a dictionary reasonable if it contains values that are likely to be observed in practice. However, this is not a sufficient restriction for *soundness*, i.e. that results derived from considering the dictionary as the superset of the accessible information must include all correct values. So we have two forms of uncertainty: analytic uncertainty with respect to the dictionary, and empirical uncertainty with respect to the resolution of the correct dictionary entry from available information. The former occurs in cases where the true set of values may not be a strict subset of the considered values, and the latter occurs where information is lost.

The derivation of a value from accessible infromation is a *guess* and must be subject to some form *check*, or *oracle* to determine whether, in fact, it is a correct derivation. However, when analyzing real systems, an oracle is often unavailable, and so we must rely on the *likelihood* of a guess being correct.

2.2.2 Likelihood, Dependence, and Independence

Likelihood is is a defined function measuring the probability of a given value being selected for a random variable. In general, whatever the likelihood function, it is dependent on certain *regularity conditions*, a set of assumptions which imply that the probability given by the function is correct. A *uniform* distribution over a set of values implies that the likelihood of any value is equal to the inverse of the number of values. An *empirical* distribution over a set of values implies that the likelihood of any value is equal to the frequency of that value in the set.

The likelihood of a guess being correct is dependent on the amount of information available to the guesser, and is informed by the method of analysis. We therefore differentiate between *dependent* and *independent* information. Two pieces of information are *dependent* if the presence of one piece of information affects the likelihood of the other. Two pieces of information are *independent* if the presence of one piece of information does not affect the likelihood of the other.

The goal of analysis and emulation is often to maximize the likelihood of correct guesses, ideally to remove guesswork altogether, and relies on the identification and explication of dependencies between information. It is possible to quantify the capacity of an emulation by considering the accessible information present in the emulated copy in contrast to the ground-truth system. A perfect simulation would perfectly disambiguate any feature that can be disambiguated in the ground-truth system, and so the accessible

information in the emulated copy would be equal to the accessible information in the ground-truth system, and a poor emulation would disambiguate nothing (be unrelated to the original system). The capacity of an emulation is proportional to our capacity for analyzing the ground-truth system.

Chapter 3: Emulation Systems

To analyze a system, we must effectively represent its behavior for some case, or set of cases, of interest. This dissertation is concerned with the emulation of systems or the effective duplication of a system for the purpose of identifying flaws, particularly with respect to security. In Chapter 4, we will consider the process of rehosting a system to a different domain in the context of hardware and instruction set emulators, which often allow complex embedded and cyberphysical systems to be analyzed on consumer desktop computers, e.g. laptops, and discuss dynamic analysis, a form of simulation which involves instrumentation of the system under analysis, in this context. Chapter 6 and Chapter 5 will discuss consider high-level emulators, copies of a system in a more abstract domain, and function extraction, strategies allowing researchers to dissect complex hardware, firmware, and software systems into testable subcomponents. Both Chapter 4 and Chapter 4 consider the use of symbolic execution and abstract interpretation as a mechanism for emulation, as these techniques explore multiple execution paths and thus implicitly emulate any single, particular execution trace as long as their lifting, semantics, and search strategies are able to identify it. Every one of these discussed formalisms is well-known and each has several systems that implement the technique or strategy for a variety of real-world use cases [52, 53, 54, 55, 56, 57, **?**, 58]. This dissertation may be the first to organize these approaches as forms of emulation—detailed literature reviews are included alongside Chapters 4, 5, and 6.

In this chapter, we will introduce four different forms of emulation used in the analysis of complex systems. For each form, we will identify contemporary systems that implement the approach, define its nature and limitations, and provide examples.

3.1 PERIPHERAL EMULATORS AND INSTRUCTION SET SIMULATORS

Embedded systems have different requirements than general purpose computers, and typically interact with specialized peripheral devices, such as actuators, and often run on different architectures designed for low power or high reliability usage (though the choice of ISA may also be due to age, cost costraints, and similar root causes). We next define the core elements of emulators designed to duplicate the operation of embedded systems on general purpose computers [59].

Microarchitectural Semantics Semantics, for example, implies the content of the 12,940 pages of the Architecture Reference Manual [60], which describes the behavior of the ARMv8-A ISA, and the representation of the processor's state.

Base semantics for a microarchitecture consists of the behavior and specification of *instructions* which are capable of being executed, the *state* these instructions can affect, and *peripheral* interfaces, such as memory mapped I/O. Instructions and state capture of the security model of the architecture, such as exception handling and memory model capabilities (e.g. read-only access). Peripheral interfaces define the capacity for the microarchitecture to interact with its environment, though not the behavior of the peripherals themselves (discussed in Sec. 3.1). The entire semantics of the microarchitecture is typically captured in a *specification language* for establishing the relationships between components.

Complex microarchitectural semantics may also be established through *microprogramming*. Microprogramming is a technique for implementing the behavior of an instruction by executing a sequence of simpler instructions. The execution of several microcode instructions effectively emulates the behavior of a single instruction. As a result of microprogramming and similar complex semantics, modeling the behavior of a processor is a highly complex task and provides continuous challenge to development of effective simulations for embedded systems.

Instruction Set Simulation An instruction set simulator (ISS) simulates the execution of instructions for a given ISA, typically by maintaining the state of the processor in its memory.

There are two primary approaches to implementing an ISS. The first is to implement the behavior of each instruction in the ISA in a programming language, such as C, and to execute the instructions in the simulator by calling the appropriate function, maintaining the processor state in variables. The second is to translate the effective behavior of the instructions into equivalent instructions on the host system, and to execute the translated instructions. The first approach is easier to implement but the second approach is typically faster, as it avoids the overhead of function calls and the need to maintain the processor state in variables. Modern architectures also increasingly support *virtual machine* (VM) extensions, which allow the host system to adopt the second approach while maintaining security.

For many architectures there exist semantics that do not exist outside of reference documentation and the processor itself. For these elements, the simulator must either ignore these semantics or emulate them through translation helper routines. These helper

routines and the translation rules for the simulator form an intermediate representation of the target's microarchtectural semantics. The simulator may then be reused and leveraged to emulate a number of systems that are based on the represented ISA without necessitating ownership of a chip implementing the ISA.

Example 3.1.1 (QEMU)**.**

QEMU is a general purpose emulator that supports a wide variety of ISAs and hardware peripherals [52]. QEMU is both a full system emulator, meaning that it is capable of emulating the behavior of an entire system, including peripherals, and capable of emulating the behavior of a processor in a virtual machine, which allows it to leverage the virtual machine extensions of the host system to accelerate the emulation of the target processor. In cases where native execution is not possible, QEMU will emulate the instruction by executing a sequence of Tiny Code Generator (TCG) instructions operating over a C struct that stores the target architecture's microarchitectural state. For example, the following PowerPC instruction:

```
0xfff0010c:   stw     r0,4(r1)
```

translates into the following TCG instructions (where qemu_st_i32 refers to the *guest* memory):

```
0xfff0010c:   movi_i32     tmp1,$0x4
              add_i32      tmp0,r1,tmp1
              qemu_st_i32  r0,tmp0,beul,3
```

TCG instructions are a RISC-like instruction set that is designed to be easy to translate to the host system's ISA, and serve as an intermediate representation for the emulated system's machine code [61]. Due to the complexity of certain microarchitectural semantics, QEMU also contains hundreds of TCG helper methods, which operate over the same C structure for the target architecture using C code. These helper functions are triggered by failures in the translation process, and because not all of them are supported or correct, QEMU serves to emulate targets rather than reproduce them.

Note that the emulation process itself is composed of a larger execution loop, which is responsible for fetching instructions, translating them to TCG instructions, executing them, and modeling the behavior of hardware peripherals and interactions between multiple processors with respect to a given ISA's interfaces.

Hardware Peripherals A *peripheral* emulator models the behavior of hardware peripherals in order to better emulate the behavior of the overall system. As a result, hardware peripherals must be represented either by connecting the emulator to the actual hardware, or by reproducing the peripheral in software. As the space of potential peripherals is theoretically infinite, it becomes more essential to explicate the interfaces by which information is passed from hardware to the emulator.

Hardware peripherals are typically connected to the processor through a *bus*, which is a shared communication channel that allows the processor to communicate with the peripheral. Interfaces to a given bus are varied and dependent on the specific microarchitecture, as bus is a catch-all term covering hardware, software, and involved protocols. For example, a processor may communicate with a peripheral through a memory-mapped I/O interface, which allows the peripheral to be accessed through the processor's memory interface, or a peripheral may communicate to the processor via an interrupt controller, which signals an exception to the processor.

A peripheral emulator must emulate the behavior of the peripheral and the behavioral portions of the processor's interface to the bus not covered by the ISS. This is nontrivial as it can require deducing specific information about the peripheral layout and memory model of the embedded system under analysis, such as the I/O addresses of a given peripheral. This information is not necessarily available for proprietary embedded systems and in these cases must be inferred from firmware code or other sources.[1]

Instrumentation For the purposes of this thesis, we are interested in *instrumentation* of the ISS and peripheral emulator to collect information about the execution of the target system (Tab. 3.1). The amount of instrumentation possible is dependent upon both the technique used to analyze the system and the completeness of the emulation.

Because these emulators have complete insight into the behavior of hardware, it is generally possible to *actively* instrument them in order to collect information about execution. Each instruction translation can be modified to collect information about various flows of execution, such as transfers of control or data. It is also possible to inject artificial instructions and data into the emulated system, in order to explore the system's behaviors during certain peripheral interactions or when receiving particular inputs. Extending this definition, it is also possible to execute the system from an entirely synthetic state in order to model specific behavioral traces. If the simulator is sufficient, then it is also

[1]For example, Systems-on-Chip (SoCs) may be programmed with specific fuse values and flashed with read-only memory to indicate information about the peripheral model and some modern operating systems use specification files, e.g. device tree blobs, to indicate the layout of memory.

possible to use active instrumentation to secure otherwise insecure operations, such as by adding bounds checks to memory accesses or by adding checks to ensure that the system is not in an invalid state.

Technique	**Description**
Instruction Injection	Modifies the translation of or injects instructions
Data Injection	Introduces data into the system at some location
Synthetic Execution	Executes the system from a synthetic state
Secure Policy Enforcement	Adds security checks to the emulated system
Monitoring Frameworks	Monitors the execution trace of the emulated system
Interface as Subsystem	Incorporates the emulator into a larger system

Table 3.1: Active and passive instrumentations in instruction set and peripheral emulators.

Passive instrumentation is also possible, and active techniques are often combined with passive techniques to improve analysis. This involves implementing monitoring frameworks which operate on the execution trace of the system and temporal reasoning operations for interpreting this trace. These frameworks can provide performance benchmarking infromation but also significantly aid in the reverse engineering of the target system by indicating key information, such as explored paths in a symbol-stripped binary. Passive instrumentation may also use the emulator as a black box and incorporating it into a larger system such as an enviromental simulator, to evaluate the dynamics of the larger system.

3.2 DYNAMIC ANALYSIS

Instrumentation is one method of *dynamic analysis*, which analyzes the behavior of a system or emulation during execution. To be concrete, the system must be running for it to be considered dynamic, and the analysis must be performed during execution, and the forms used for this dissertation include live binary rewriting, fuzzing, the use of debugger, and passive monitoring.

Live Binary Rewriting Sometimes it is possible to add self-modifying instructions to a binary in order to change its behavior as it executes. For a given system, different forms of rewriting and instrumentation are possible, and most include the use of *shim* code injected between the operations of other code for instrumenting the state of the system. The shim itself can be implemented in a variety of ways, such as by using a *trampoline* which maintains the setup and teardown of the shim's state during instrumentation.

As a common example, the length of a call instruction on i386 (x86) is a 5-byte instruction, consisting of one byte for the opcode and four bytes for the address of the function to call. As long as the four byte offset is sufficiently small, the actual call target location information only requires the middle three bytes of the instruction. In these cases, these three bytes can be shifted over by one, and the call instruction may be modified to a 2-byte interrupt instruction, which transfers control to an interrupt based trampoline for the performance of instrumentation and, reads the remaining three bytes to determine the correct return address, and then self-modifies the shim to replicate the control transfer using the three byte offset (Fig 3.1). This same technique was used to develop exploits for the CMU-900 in Chapter 4.

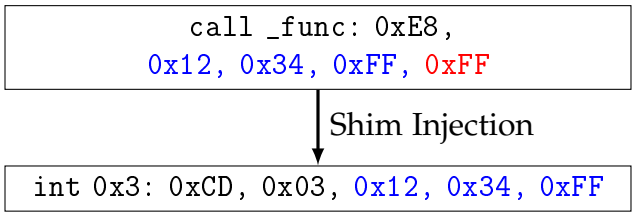


Figure 3.1: Example of binary rewriting.

Fuzzing Fuzzing is a technique for generating inputs to a system in order to explore its behavior. In general, fuzzing must be performed on an emulation rather than the system itself, as it is necessary to reset a correct initial state for fuzzing, and many crashes or unexplored behaviors may damage the real system. For the discovery of bugs, fuzzing is effectively an intelligent dictionary attack which must infer the right inputs to test from a large number (potentially exponential) of possible inputs. Effective fuzzing also requires careful definitions of *exit conditions*, which indicate when a given input has not found any interesting path, as exploring paths which "hang" decreases overall analysis performance. The most common fuzzers in use today are variations of American Fuzzy Lop (AFL) [62], which provides a general framework and API for building more advanced fuzzers attached to emulators.

Debugging Debuggers are a common tool for analyzing the behavior of a system, and are often used in conjunction with other techniques. Similar to a shim, debuggers work by halting the execution of a system at a given point, allowing the user to inspect or modify the system's state, and then resuming execution. These inspection routines may also be automated, and can be used to collect information about the system's state during execution, similar to instrumentation. For this purpose, many emulators, like QEMU, include a debugging (gdb) server which allows the analyst to connect to the emulator and inspect the system using common mechanisms, e.g. reading memory values.

3.3 HIGH LEVEL EMULATORS

While the above techniques are often necessary for the analysis of opaque embedded systems with symbol-stripped firmware, in many cases binaries also originate from standard consumer hardware. For these binaries *functional* or *high level* emulation (HLE) is appropriate—emulation that does not require the rehosting of system components, and directly operates on program representations. Standard executables like Microsoft Word are distributed as a binary in a common format with some symbols intact, and emulation can run on standard laptop hardware as the system does not need significant rehosting. These binaries can be instrumented, translated, or analyzed using a number of tools, including symbolic reasoning frameworks and standard tools from formal methods. However, these emulators can also be generated using *function extraction*, *summarization*, and *decompilation* of embedded system binaries.

A HLE can make the model of the system more effective and useful while sacrificing authenticity. These emulators abstract the lower level components or software of the system either by assuming they are already present on the host or by ignoring them altogether. One benefit of this approach is that it can more easily address high-level correctness concerns, such as the reachability of certain system states, as it does not require a thorough analysis of the system's subcomponents and semantics. This approach is also independent of detailed hardware specifications but is less flexible, as the high-level emulator defines a limited API for the analysis of the system under consideration. Moreover, a given high-level emulator will be standardized to a particular interpretive domain, e.g. a dynamical or hybrid system interpretation, and does not simulate the behavior of the system under study, which may be possible to model using mathematics but is only exactly specified by the system's implementation in itself.

As more complex, widely used systems and software become prevalent, HLEs become

critical. Supporting the semantics of each detail for a complex system is often infeasible or simply wasteful, as the features are already present on the host for the emulation or not necessary for the analyst's goals. Low-level emulations are also becoming easier to generate as systems move to single, well-understood architectures like ARM.

Function Extraction HLEs often involve some method of *function extraction* for a given component from a complex system, which allows a portion of the system to be emulated in isolation from other components. Extraction can involve a range of techniques, including program slicing, abstract interpretation, lifting and *summarization*. A summarization of a function f is given by the following definition [63]:

Definition 3.3.1 (Function Summarization)**.** *Let f be a function, v a bound on the number of unrolled loops and recursive calls, R_v^f a set of tuples of computations in f over v, $\mathbb{D}$ a domain function mapping from inputs to outputs of f, then S s.t. $R_v^f \subseteq S \subseteq \mathbb{D}$ is a summary of f.*

This definition in itself is limiting as it only considers a strict subset of the map from the inputs to the outputs of the computation f, and does not consider the relations that exist between tuples in R_v^f. Thus it becomes appropriate to consider a more general definition of *logical translation*, which uses propositions to extract system components.

Definition 3.3.2 (Logical Translation)**.** *Let f and f' be functions, $P = p_1, p_2, \ldots$ possible execution paths through f, $\Phi = \phi_1.\phi_2.\ldots.\phi_n$ a set of statements in natural deduction for resolving the relationships between symbols in one computational domain function, $\mathbb{D}$, mapping from inputs to outputs of f, to another domain function, $\mathbb{D}'$, mapping from inputs to outputs of f', $A^f_{\Phi(P)}$ a set of tuples of computations resulting from the application of Φ to f over P, then S s.t. $A^f_{\Phi(P)} \subseteq S \subseteq \mathbb{D}'$ is a logical translation of p.*

This is formalized via natural deduction, but often the specific translation process is not fully proven for real-world systems and instead adopts *heurisitics* or short-cuts which are assumed to be correct in most cases. An example application of logical translation in this dissertation is the extraction of glyph positioning schemes from PDF document creation software by deducing the C representation of important computations from data-flow analysis (dynamic instrumentation) of machine code operations.

Regardless of the method used, the resulting extracted function may be used as an emulator to verify and understand the behavior of the system, but may be missing non-extracted interfaces. There are several methods of addressing this, such as replacing these intefaces with logical summaries, as in the HiFrog [64] and later UpProver [65] systems, or through the use of fuzzing. Not extracting hardware peripheral semantics also

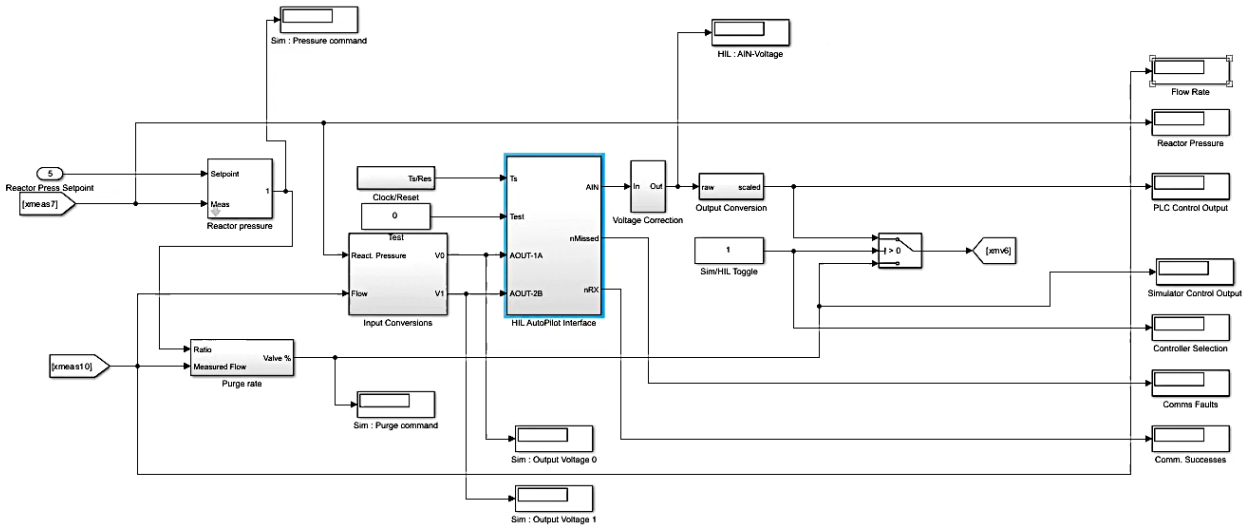


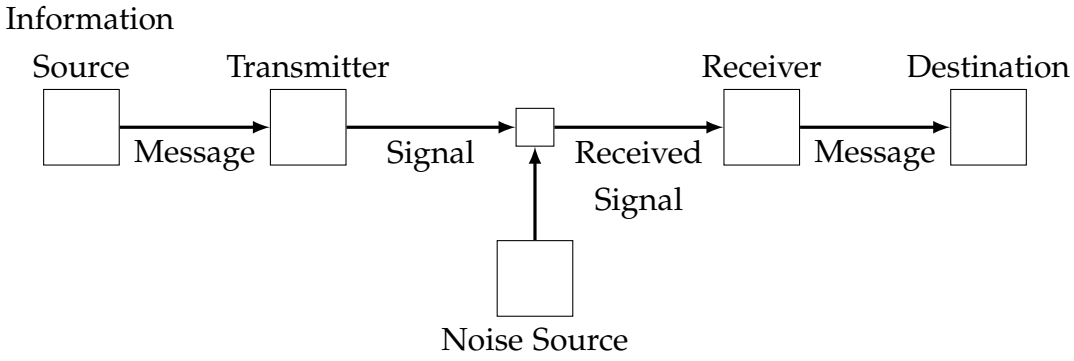
Figure 3.2: Environmental model emulation inputs and outputs for a chemical plant communicating to a physical Programmable Logic Controller (the blue block) in Simulink.

makes firmware rehosting systems like P^{2}IM [23] implicit forms of function extraction of software from a hardware-software system, as evidenced by recent work on incomplete information in firmware specifications [33].

Environmental Modeling and Communication Channels Emulation can extend to the modeling of the *environment* of a system, and oftentimes the extracted function of a single component may not be sufficient to understand the overall real-world behavior or security of a system. Similar to peripheral modeling, some information about the system's environment may not even be available without additional external analysis. Non-computational environmental modeling is constrained to three forms of specification, captured by common toolkits like Matlab Simulink, used in this dissertation: equations, random variables, and logic. Models of computation and stateful systems in the environment are also possible and captured by the subdomain of *hybrid systems* which unify discrete and continuous state transitions [66].

Figure 3.2 depicts part of the environmental model for a Tenessee Eastman chemical plant used in this dissertation. Regardless of the interconnections and components present in a given model, it is noteable that the emulation of a given subcomponent is entirely determined on what the environmental model can *communicate* to the subcomponent. This concept generalizes across emulation and information theory, captured in by the concept of a *channel*, which consists of an *information source*, *transmitter*, *noise source*,

receiver, and *destination* [67]:



Noise constitutes any distortion of the signal (e.g. loss of information). The *channel capacity* is the maximum amount of information that can be sent through the channel, and is given by the following equation:

$$C = \max_{p(x)} I(X;Y) \tag{3.1}$$

where $p(x)$ is the probability of a given input x to the channel, and $I(X;Y)$ is the *mutual information* between the input X and the output Y of the channel. Mutual information is defined as:

$$I(X;Y) = \sum_{y \in Y} \sum_{x \in X} p(x,y) \log \frac{p(x,y)}{p(x)p(y)} \tag{3.2}$$

where $p(x,y)$ is the joint probability of x and y occurring, and $p(x)$ and $p(y)$ are the marginal probabilities of x and y occurring. A channel is considered to be *lossless* if the mutual information between the input and output is equal to the entropy of the input, and *lossy* otherwise. We relate this measure of mutual information to the capacity for an emulation to determine unknown qualities of a system in Chapter 5. Shannon's information channel is often also generalized into an *information flow* which moves between different privilege domains and quantifies security [68, 69].

3.4 ENCODING, DECODING, AND MODELING

The formalization of *codes*, systems of rules for the conversion of information, can unify many challenges presented by hardware emulators of Section 3.1, dynamic analysis of Section 3.2, and high level emulation of Section 3.3. Traditional coding theory is concerned with the design of codes that can be used for the reliable and efficient transmission of

information across a noisy channel [70, 71, 72]. Standard analyses of codes involve the removal of redundancies and detection of errors and a study of communication channels. One can also understand emulation as a decoding of a communicated information about the system and an encoding of the system's behavior. Thus, the efficacy of an emulation is determined by its ability to decode the original system and its ability to encode relevant information, typically thought of as a minimum description length (MDL) or Kolmogorov Complexity.

Definition 3.4.1 (Minimum Description Length)**.** *The length C(s) of the shortest binary string s that is delimited by special marks and that can compute x on the Univeral Turing Machine and then halt.*

This definition has several limitations:

1. It is not computable [73]. Informally, to determine an MDL we must execute every program and collect all that stop and compute x, choosing the one that has the smallest size. However, from the halting problem, we cannot decide if a program stops or not.

2. There are many programs that compute x and halt, and emulation is not strictly concerned with the computation of an *optimal* encoding, but an effective encoding in a high-dimensional space.

3. The definition does not provide insight into noise present in the channel, and information loss is common in the analysis of real-world systems. Equivalently, it does not provide insight into the encoding protocol of the information and representation of the system (assumptions made regarding the optimal description).

MDL presents an important insight into the *relative* complexity of system encodings through the measure $C(s)$ which, when generalized, can determine which of two emulated representations is more optimal. This is typically evaluated using the results of the dynamic analyses presented in Section 3.2. There also exist standard encoding methods for system descriptions, encapsulated in prevalent techniques.

Taint tracking is one such standard method of system description, used to encode the flow of information. For every bit of information in a system, a bit of taint is associated with it, and set to 1 if the information is tainted, and 0 otherwise. Then, for every operation in the system, the taint of the output is determined by the taint of the inputs and the operation. If input bits are tainted and these bits decide some bits output, then these

Algorithm 1 Backpropagation

```
1: procedure BACKPROP(X, Y, W)
2:     for l ← L − 1 to 1 do // Each layer
3:         for i ← 1 to n do // Each neuron
4:             δ_i^l ← (Σ_{j∈L_i} w_ij^{l+1} δ_j^{l+1}) * f'(z_i^l)
5:         end for
6:     end for
7: end procedure
```

output bits will tainted. The resulting output bits encode how certain bits of information are used throughout the system.

Standard encodings such as taint and associated emulators can be used for a variety of purposes. In Chapter 4, we will use taint tracking to bootstrap the operation of a symbolic execution engine within an instruction set simulator. For general function extraction, a standard encoding is an *execution trace*, which can be recorded and then "replayed" to emulate the behavior of a system with the possibility of *time-travel debugging*, a form of source-to-sink information-flow analysis capable of extracting the specific operations used to derive a variable.[2] Execution traces are written by some *interpretation* of the system given by the emulator, a transformation of information which provides that information with a concrete meaning in some context.

A given trace will be incorrect if the emulator does not capture the semantics of the system which are captured by a "correct interpretation", e.g. execution on the target microarchitecture, in the trace's output encoding (Section 3.1). Execution traces therefore define an ordering on emulations, where the more optimal emulation is the one which most closely captures the desired semantics of the ground-truth system in its output.

Modeling a system is the generation translation of the system's original encoding which is still executable either in the original context or in an emulator. Formally, a model of a system will preserve some property of the system, such as its behavior or its structure, to some degree of error ϵ. Given appropriate assumptions on the intended semantics of the system hold, the *correctness* of a model may be determined by a proof demonstrating the outputs of the model are within ϵ of the outputs of the original system. This problem is also refered to as the *completeness* of an interpretation [75].

Example 3.4.1 (Artificial Neural Network)**.** *Consider a system with a subcomponent implementing a single backpropogation step of an artificaial neural network (ANN), provided in the*

[2]We scripted the Windows debugger time-travel debugging feature [74] in order to peform function extraction on the glyph positioning models for the purposes of breaking redactions in Chapter 5.

```
/* Code ommitted for space */
else {
  /* Code ommitted for space */
  do {
    pdVar15 = pdVar7 + 1;
    uVar22 = __aeabi_dmul(
      *(undef4 *)pdVar7,
      *(undef4 *)((int)pdVar7 + 4),
      *(undef4 *)(pdVar17 + 1),
      *(undef4 *)((int)pdVar17 + 0xc));
    pdVar17 = pdVar17 + iVar14;
    uVar21 = __aeabi_dadd(
      (int)uVar21,
      (int)((ulonglong)uVar21 >> 0x20),
      (int)uVar22,
      (int)((ulonglong)uVar22 >> 0x20));
    pdVar7 = pdVar15;
  } while (pdVar15 != pdVar11);
}
uVar10 = *(undef4 *)(local_78 + 1);
uVar13 = *(undef4 *)((int)local_78 + 0xc);
uVar22 = __aeabi_dsub(0,0x3ff00000,uVar10,uVar13);
uVar22 = __aeabi_dmul(
  (int)uVar22, (int)((ulonglong)uVar22 >> 0x20),
  uVar10, uVar13);
dVar23 = (double)__aeabi_dmul(
  (int)uVar22, (int)((ulonglong)uVar22 >> 0x20),
  (int)uVar21, (int)((ulonglong)uVar21 >> 0x20));
/* Code ommitted for space */
```

Figure 3.3: Ghidra decompiled code for the sum operation of Algorithm 1.

```
tmp = z3.Function('f', [z3.FPSort(11,53)]*3)( z3.fpBVToFP(
   z3.Concat(r1, r0), z3.FPSort(11, 53)), z3.fpBVToFP(
   z3.Concat(r3, r2), z3.FPSort(11, 53)))
r0 = z3.Extract(31, 0, z3.fpToIEEEBV(tmp))
r1 = z3.Extract(63, 32, z3.fpToIEEEBV(tmp))
```

Figure 3.4: Example model of a ARM software floating point add as an uninterpreted function.

high-level encoding of Algorithm 1. This standard model of the ANN could be "executed" by a human reader, but the form of encoding executed by a computer is the low-level encoding of the system, which is a sequence of instructions given in machine code or interpreted from a high-level representation. Fig. 3.3. depicts one such high-level representation, the output of this routine decompiled *from machine code into a pseudo-C representation by Ghidra using several of the techniques mentioned in this Chapter. This code, which represents a subcomponent of a larger system, could alternatively be modeled by a symbolic executor, such as angr. Upon reaching a function like __aeabi_dadd, traditional symbolic execution will enter into the function, interpret it, and return a complex bitvector operation.*

To provide some notion of complexity, one software floating point multiply we studied, for example, had 153 lines of pseudo-C, 116 lines of decompiled assembly, and 15 branches, but a given emulator may interpret only a subset of these instructions. Extending this notion of emulation to the absolute limit, it is also possible to encode a system in such a manner that all *internal semantic detail is lost and only the input and output characteristics remain.*

We extracted one ARM software floating point value add taken from a real implementation of the above ANN code for a specific compiler and microarchtecture. This set of instructions had inputs split across four registers and outputs in two registers. Represented as an uninterpreted function, preserving no semantics of the add operation itself, in a format that may be operated on by the Z3 theorem prover [76][3] results in the encoding provided in Fig. 3.4. The specific encoding of a system is therefore highly flexibile, justifying the need for empirical evidence and methods of evaluation when adopting strategies in these subject areas.

[3]We split the representation into three assignments for simplicity.

Chapter 4: Targeted Firmware Rehosting

Embedded systems pose a challenge for emulation because their code expects to interact with specialized on-chip and off-chip peripherals, such as general-purpose I/O (GPIO) ports, sensors, and communication interfaces. In this dissertation, we consider this specific challenge in the context of *rehosted* emulation. Introduced in Chapter 3, rehosting refers to the decoding of information from a system specification into a set of emulated systems capable of executing that system in a different domain, e.g. a laptop computer. The execution environment must emulate these devices with sufficient fidelity to ensure that observed behavior accurately mimics the target system running on hardware. However, because of the large variety of peripheral devices, most are not modeled by the execution environment, creating a considerable blind spot for our most powerful analysis techniques. Indeed, there may be no documentation at all about a target system, which makes building a complete emulator for it nearly impossible.

In many cases, however, the code of interest to the system analyst is not the code that interacts with peripherals. While peripherals cannot be ignored completely—hardware initialization must appear successful for the system to boot successfully—correct behavior of all devices may not be necessary. This chapter will focus on the rehosting approach, which attempts to infer partial but effective models of these missing hardware devices, and Chapter 6 will look at an extraction based approach to this same problem. For example, an analyst interested in how a target responds to network traffic may not require the execution environment to faithfully model all aspects of the system's GPIO ports or other communication interfaces.

The subject of this chapter is Jetset, a system that performs *targeted rehosting* of firmware—it automatically infers the expected behavior of embedded system peripherals using only

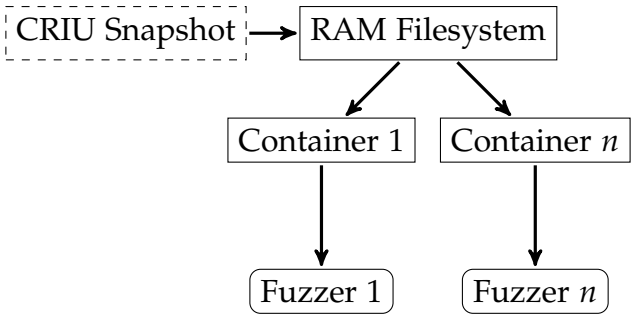


Figure 4.1: The CRIU and Docker-based QEMU fuzzing system used to evaluate Jetset's rehosting of emulated systems.

its firmware and then synthesizes a model of the peripherals sufficient to boot to security-critical code of interest. The synthesized peripheral model can then be used in an emulator—in our evaluation we used QEMU [52]—to emulate the hardware environment. An analyst can then use her tool of choice to interact with the firmware. For example, a vulnerability analyst can use Jetset to fuzz-test the system to see how it responds to malformed or otherwise malicious input. More advanced dynamic analyses, like symbolic execution, are also available to the analyst.

Jetset infers the values that need to be read from peripheral devices needed for the program to reach an analyst-specified *goal address*. For example, on the Raspberry Pi target used in our evaluation, our goal is to reach the address where the code jumps to user space. Our key insight is that firmware code interacting with a peripheral device implicitly encodes how the device must behave for the system to boot. Jetset uses symbolic execution of the firmware—specifically the angr framework [**?**]—to infer data returned from devices.

The input to Jetset is the executable firmware image, the firmware entry point (where to start execution), a goal address (the address we want to reach in execution), and a memory layout specifying which parts of the address space represent RAM and which represent memory-mapped I/O. Jetset only requires emulation support for the CPU architecture; it does not require any special hardware, and does not use the underlying hardware device.

To evaluate Jetset, we use it to infer and instantiate peripheral devices for thirteen targets: an aircraft Communication Management Unit (AMD 486-based system) used on the Boeing 737, Linux on a Raspberry Pi 2 (ARM-based SoC board), the first-stage bootloader on a BeagleBoard-xM (ARM-based SoC board), a SEL-751 Feeder Protection Relay (Motorola ColdFire-based system), and the 9 publicly available real-world targets from prior comparable work[23]. These targets are diverse—they come from 3 different architectures, 5 different operating systems (as well as 3 different bare metal systems), and several different application domains. For each target, Jetset inferred the behavior of its peripherals needed for the firmware to complete its boot sequence,and produced C code suitable for use with QEMU that simulates the inferred devices. We then run the firmware in QEMU configured for the target CPU architecture using our synthetic peripherals to complete the configuration.

For two of our targets, we confirm that the synthesized devices work correctly by comparing the emulated system against a reference. For the Raspberry Pi 2, we compare the emulated behavior of the system to its behavior on actual hardware. We used a novel framework capable of snapshotting the QEMU device emulator and instrumenting this

with the AFL fuzzer [62] to fuzz-test the Linux kernel system call interface, obtaining the same results both in QEMU and on the actual hardware. For the Communication Management Unit (CMU), we use a high-fidelity QEMU implementation of the system, including its most important peripherals for comparison. We produced this implementation by manually reverse-engineering the CMU as our reference. We used our fuzzer to test the system call interface of the underlying OS on both the reference and synthesized implementations to confirm we observe the same behavior on both. Although finding vulnerabilities was not the goal of this testing, we nevertheless identified a previously unknown privilege escalation vulnerability in the VRTX kernel used by the CMU (Section 4.3.2). This chapter will also discuss other exploits developed using this emulator analysis and disclosed to UTC aerospace: first, a local code execution method based upon the data loader protocol, and ACARS messages which are capable of disabling a running CMU remotely.

4.1 RELATED WORK ON REHOSTING

Due to the complex nature of firmware and the heterogeneity of the hardware it interacts with, security testing and analysis of firmware is a difficult problem [77, 20]. Different techniques to test and analyze firmware vary both in their goal (e.g., finding bugs, full rehosting, or partial rehosting), as well as the assumptions that they make about the firmware they analyze. For example, a testing technique may only analyze firmware using a particular operating system [78], or may assume that auxillary information about the firmware is available (e.g., firmware-hardware I/O traces [79]) to improve results. The use case of Jetset—partial rehosting using only the firmware itself and no auxillary information—is most similar to other rehosting techniques, however, for completeness, we outline other approaches to analyzing firmware below.

Firmware testing and analysis Approaches have been developed to test firmware without attempting to create a stand-alone emulator for the hardware.

Symdrive [80] is a symbolic testing framework for Linux device drivers. Symdrive takes as input the C code for the Linux drivers and attempts to find program paths that violate user written assertions. Symdrive is able to uncover numerous bugs in Linux device drivers; however, it requires source code and is Linux specific.

FIE [81] is a symbolic execution framework that targets firmware for the MSP430 family of microprocessors. FIE takes as input a piece of firmware, a memory map (that denotes

which regions are RAM, ROM, MMIO, etc), and an interrupt specification which describes all locations where interrupts could be fired. FIE is designed to analyze all firmware execution paths, which, while effective for the simpler MSP430 microcontroller firmware, is not feasible for more complex firmware like the Raspberry Pi's Linux kernel. For this complex firmware, a more targeted approach such as Jetset's is needed. Furthermore, FIE requires the source code for the firmware—this is how it adds its symbolic execution instrumentation—and it is therefore unsuitable for our needs.

Revnic [82] is a system for symbolically executing driver firmware and reverse engineering its functionality. Revnic takes as input a driver binary, a driver template describing the high level functionality of the driver, and domain specific knowledge about the OS of the driver, and produces source code for the driver. Revnic requires knowledge of the underlying operating system, and requires that the user provide detailed device templates that outline the functionality of the device, and it is therefore unsuitable for the problem of firmware-only emulation.

FirmUSB [83] is a USB-specific symbolic execution framework for analyzing USB microcontroller firmware. FirmUSB takes as input a USB firmware image, and uses domain specific analyses to identify malicious behavior by the USB device. For example, FirmUSB can detect if a device claiming to be a USB keyboard is injecting keys that have not been pressed by looking for USB specific information flows.

Hardware-in-the-loop emulation Another method of approaching the problem of analyzing firmware is to attach a software emulator running the firmware to the physical hardware, forwarding I/O between the emulator and the firmware. Avatar [84] is a dynamic analysis framework for embedded systems that does exactly this. Other tools Surrogates [85] and Prospect [86] build on this hardware-in-the-loop approach.

This technique provides the highest fidelity emulation since the emulator directly interacts with the physical hardware; however, use of this technique is contingent on continuous access to the hardware, which is not always possible since hardware (like that used in avionics) may be difficult or impossible to obtain.

Full firmware rehosting Full rehosting is a technique which attempts to construct a fully featured, high-fidelity emulator from a piece of firmware and auxillary information about the SoC or firmware.

Firmadyne [78] is a platform for automated dynamic analysis of Linux-based embedded systems. Firmadyne takes as input a piece of firmware running the Linux kernel, and executes user-space code for the firmware, emulating the common Linux peripherals.

Similarly, Costin et al. [87, 88] extract and rehost the embedded system's filesystem in their own analysis environment to analyze network-facing code. Because the code of interest to an embedded system security analyst is often the user-space, network-facing code, Firmadyne and Costin et al.'s tool are well-suited for this scenario.

Pretender [79] rehosts firmware by recording the interactions between the physical hardware and the firmware. It then uses a machine learning engine to learn a stateful model for peripheral behavior and creates an emulator from this model. Similar to Avatar, Pretender takes as input the firmware, and a connection to the physical hardware, and creates an emulation environment; however, unlike Avatar and related tools, Pretender can fully migrate the firmware to a virtualized environment, and does not require persistent access to the hardware.

HALucinator [24] is a firmware rehosting tool that uses hueristics to locate the code belonging to the hardware abstraction layer (a vendor-provided API for interacting with the hardware) in the firmware and replaces it with manually created handlers. HALucinator takes as input firmware, and the HAL the firmware uses, and produces a fully featured emulation environment for the firmware.

Previous rehosting techniques have relied on auxillary information to infer the behavior of the hardware environment. While this results in a more complete emulator, this auxillary information is not always available—most of our evaluation subjects had none. Furthermore, security analysis is often concerned with only a particular software component of the firmware, (e.g., the network traffic or the file system code) and may not need a fully featured emulator.

Partial rehosting Partial rehosting, as opposed to full rehosting, attempts to create an emulator from the firmware only, with no auxillary information about the peripherals. However, the emulators produced by partial rehosting are not complete—they are not guaranteed to implement all peripherals for the firmware, only what they can infer. This is the point in the design space that Jetset occupies. There is one other notable system that implements partial rehosting, P^{2}IM.

P^{2}IM [23] does both fuzzing and partial rehosting based on the peripheral model that it infers from the fuzzing stage. It takes as input the target firmware and its memory map, and fuzzes the firmware code by channeling input from an off-the-shelf fuzzer like AFL to the peripherals. It then analyzes the device access patterns exercised during this fuzzing pass to infer details about the memory-mapped IO interactions between the firmware and peripheral devices, and executes the firmware without crashing.

There are two key differences between P^{2}IM's fuzzing-based approach, and Jetset's

directed symbolic execution-based approach. The first deifference is that unlike P^{2}IM, Jetset is *targeted*—it is designed to ignore most paths through the firmware to focus on a particular target piece of code, which allows it boot deep into large pieces of firmware. While Feng et al. showed P^{2}IM's approach is effective at fuzzing peripheral handling code and emulating microcontroller code, it is not clear whether it scales to larger firmware. Besides evaluating against all of P^{2}IM's publicly available real-world evaluation subjects, we also evaluated Jetset against four complex pieces of firmware—one of our evaluation subjects, the Raspberry Pi 2 is 450x LoC of any of P^{2}IM's evaluation subjects. We attempted to evaluate P^{2}IM on our 4 real-world firmware samples. Unfortunately, the current version of P^{2}IM only supports Cortex-M MCUs and we were unable to run it on any of our samples, including our Cortex-A7 and Cortex-A8 firmware.

The second difference is that, while fuzzing-based approaches are efficient since they use lightweight executions, they can have trouble bypassing complex checks. In Section 4.2, we provide an example of a complex numerical check that occurred when inferring the behavior of an FPGA in one of our evaluation subjects. Jetset is able to handle complex numerical checks, because it performs partial rehosting using symbolic execution.

4.2 JETSET'S INFERENCE OF HARDWARE SEMANTICS

Jetset uses symbolic execution to infer how peripheral devices must respond to reads from the firmware for execution to progress toward the goal address. It uses this inferred information to deduce and reproduce expected peripheral device functionality to boot firmware in an emulator such as QEMU. This allows analysts to boot the system in an emulator with only the firmware, and without the target's hardware or support for the peripheral devices in the emulator. To do this, Jetset requires the following information about the target embedded system (discussed briefly in Chapter 3).

- The *executable code* of the target, usually read out of program flash or extracted from a firmware update provided by a manufacturer.
- The *memory layout* of the target, specifying which regions of the address space are mapped to program memory, RAM, and device I/O registers. This information can be obtained from the datasheet of a single-chip system or from a basic analysis of the executable code. Note that Jetset does *not* need to know which devices are mapped where, only the address range used for memory-mapped I/O.

- The *entry point address* where execution begins. This is often specified in the CPU datasheet.
- The program *goal address* that the analyst wants the program to reach. For example, this can be the address of a print instruction that reports a successful system boot.

To model peripherals, Jetset uses symbolic execution to *infer* expected device behavior, and then the resulting encoding of the device behavior in the constraints returned by symbolic execution is used to *synthesize* a device suitable for use in an emulator (e.g., QEMU). Secondary to this *peripheral Modeling* phase, Jetset must also adopt a correct *search strategy* to guide symbolic execution, introduced in Chapter 2.

4.2.1 Jetset's Peripheral Modeling

In its *peripheral inference* stage, Jetset symbolically executes the firmware to infer what values should be returned by reads from device registers in order for execution to reach the firmware's goal address. In Jetset, input from devices are marked symbolic and tracked through QEMU's emulation by *taint-tracking* all Tiny Code Generator (TCG) operations and a number of QEMU's helper functions. During execution, all reads from memory-mapped I/O addresses space are therefore symbolic, while the initial contents of flash and memory are concrete.[1] Each read from a memory-mapped I/O address returns a distinct symbolic variable; that is, two reads from the same address result in two *different* symbolic values. Jetset stops when an execution path reaches the goal address, resulting in a set of constraints on values read from device registers that lead to this address.

Interrupts While executing, the firmware may also require interrupts to be serviced to reach the target. Jetset therefore periodically injects interrupts during the modeling stage. For example, the goal address for the Raspberry Pi firmware is in a different kernel thread than the entry point, so a scheduler interrupt is needed to reach the goal. From Jetset's point of view, this means that it needs to execute an interrupt service routine (ISR) to make progress. Given infinite compute resources, Jetset could explore every possible interrupt either firing or not after each instruction. However, this is impractical. Jetset exploits the fact that well-designed systems are not sensitive to the *exact* timing of interrupts and that ISRs are written to handle spurious interrupts gracefully. Jetset

[1]Note that this means Jetset *does not* infer flash devices, something which was manually modeled for the emulation during the QEMU exploit generation.

periodically injects interrupts during symbolic execution, so that each ISR is executed periodically during each execution path. If the main execution thread happens to be waiting for an ISR to update a variable, Jetset will eventually execute that ISR, and the thread can continue making progress.

Peripheral synthesis Once Jetset has reached the target (with or without interrupts), it can create a synthetic peripheral model that can be used in QEMU. The result of the inference stage is a set of constraints on values read from peripherals needed for the firmware to boot. Jetset then uses Z3 [76], the default SMT solver used by angr, to find concrete values satisfying these constraints, capturing in the firmware's expected response to device reads during execution. This allows Jetsetto construct a light-weight, concrete device model, rendering peripheral modeling a one-time cost per device.

This synthesis stage generates an I/O trace model that is sufficient to reach the goal in the emulator. The synthesized trace is partitioned by I/O address, so there is a separate trace for each memory-mapped I/O address. When Jetset reaches the end of an I/O trace for a particular address, any subsequent reads return the last value in the trace. This allows Jetset to continue past the goal address in emulation, but precludes any complex interaction with the device after the trace has ended (see Section 4.4).

The synthesized device injects interrupts in the same way as during peripheral modeling, ensuring that any necessary ISRs are executed in emulation. Interrupt timing during execution in an emulator does not need to precisely match the timing during peripheral modeling—if, during emulation, an interrupt is fired one instruction later, this will not make a difference in emulation.

4.2.2 Jetset's Search Strategy

To ensure that Jetset continues to make forward progress towards the goal address during symbolic execution, which by default attempts to explore the all executable paths, Jetset uses a distance function to guide its search. This distance function is context-sensitive [89]: it takes into account that the distance between two instructions in a program can depend on the calling contexts (i.e., the callstack) of the two instructions. Computing a context-sensitive distance function is more complicated than computing a local distance (i.e., the distance between two instructions in a single function). Whenever a decision point is encountered, Jetset chooses the shortest path to the goal location, barring a few exceptions, discussed later.

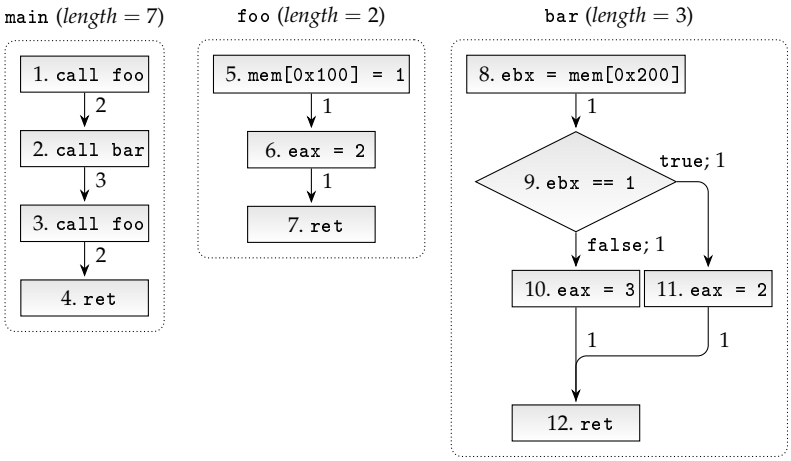


Figure 4.2: Context-sensitive distance from statement 5 (in first `foo` call) to statement 7 (of second `foo` call).

The local distance between two instructions is simply the graph distance between the two instructions in the control flow graph. For example, in Figure 4.2, the distance from statement 5 to statement 7 (both within `foo`) is 2. When computing local distances, the edges for call instructions need to be weighted based on the called function's *length*—the distance between the start of the called function and the nearest return of that function. For example, in Figure 4.2, the distance from statement 1 to statement 4 is not 3, but 7. This is because, when executing a call instruction, it is not really one instruction being executed, but every instruction until the call returns. This is further complicated, because the called function may itself call other functions. Therefore, to compute local distances for each function, Jetset first creates a callgraph of all functions in the firmware, then computes local distances for functions in topographical order. This ensures that when Jetset computes local distances for a function, it has already computed local distances for every function that function calls. But this still only gives local distances—it does not provide distances between instructions in different functions.

Computing the distances between instructions in different functions is more complicated, because functions are often called in more than one context and Jetset is only interested in *realizable paths*—paths which follow a valid call-return sequence. For example, in Figure 4.2, the distance between statement 5 (in `foo`) and statement 4 (in `main`) depends on `foo`'s calling context: if `foo` was called from statement 1, then the distance is 7, if `foo` was called from statement 3, then the distance is 2.

Jetset uses a context-sensitive distance function: it determines the distance between an instruction in one calling context—a (pc, callstack) pair—to another instruction, in another calling context. To compute this distance function, Jetset first precomputes local

distances for all functions. Then, Jetset computes the distances between instructions in different functions. To do this, Jetset takes advantage of the fact that all paths between instructions can be broken up into a sequence of returns, followed by a sequence of calls [89] (there will never be an interleaved call and return, because then that would be a local distance!). Nonlocal distances can therefore be seperated into two distances: the *callstack distance*—the distance along the sequence of returns up the callstack—and the *callchain distance*—the distance along the sequence of calls that lead to the goal address (or the goal address in a specific calling context).

Jetset precomputes all local distances, but both the callstack and callchain distances are computed lazily from the actual stack during execution (it is infeasible to precompute all callstack and callchain distances). Jetset computes the total context-sensitive distance as the sum of the callstack and callchain distances.

The callstack distance measures the distance from an instruction in one calling context to an instruction that can be reached by a sequence of returns (i.e., instructions in functions in the current callstack). To compute the callstack distance, Jetset first computes the local distance to the location of the closest `return` instruction. It continues summing the distances to each return of each function recursively up the call stack. It stops once it reaches a function that can reach the target with a set of calls (i.e., a function that is in the target's callstack):

```
# Calculate callstack distance
while function not in target_callstack:
  distance += local_distance(cur, ret)
  cur = function.returns_to
  function = stack.next_function
```

For example, suppose Jetset wanted to reach statement 7 (in the second `foo` call) from statement 5 (in the first `foo` call). The call stack distance would be 2, as that is the distance to exit from `foo` to `main`, at which point statement 7 can be reached by a set of calls.

The callchain distance measures the distance from an instruction in one calling context to an instruction that can be reached by a sequence of calls. To compute the callchain distance, Jetset first computes the local distance to the nearest `call` instruction that leads to the target. It then recursively sums the distance to each function call on the way to the target:

```
# Calculate callchain distance
while function != target_function:
  call = target_callstack.closest_call
  distance += local_distance(cur, call)
  function = call.target
  cur = function.entry
```

Suppose again that Jetset wanted to reach statement 7 (in the second `foo` call) from statement 5 (in the first `foo` call). The call chain distance would be 5: 3 to reach the second `foo` call and 2 to descend into the `foo` call to reach statement 7.

Search Exceptions However, there are many cases where due to incomplete semantic modeling and software control flow complexity, jetset is not able to identify a correct path to the current goal address using the context-sensitive CFG. For example, to cope with loops where taking the shortest path will lead to a hanging state, we *alternate* decisions, to avoid taking the same branch twice where unnecessary. For these cases, Jetset relies on using the local distance to the nearest return as a fallback distance function. The incremental CFG generation improves the quality of the CFG over time, so eventually the CFG will contain a path to the target.

Jetset's search algorithm may also guide it to a point in the program where it becomes infeasible to reach the target and it needs to terminate the current path and backtrack to a previous state. There are two different cases where this occurs. The first case where Jetset backtracks is when a system reset occurs; it is unlikely that a system reset takes place in a correct boot sequence, and backtracking on system resets allows Jetset to avoid boot loops. The second case where Jetset backtracks is when Jetset enters a statically-detectable infinite loop. It is also possible to set explicit "avoid" locations which will force Jetset to backtrack if some set of constraints are satisfied.

4.3 EVALUATING JETSET'S TARGETED REHOSTING

Jetset's evaluation was dependent upon getting several firmware images into a steady state emulation, where the emulator would run without crashing. The core targets for this evaluation were a Raspberry Pi 2, a single-board computer based on the Broadcom BCM2836 system-on-a-chip (SoC); a BeagleBoard-xM, a hobbist board, based on the Texas Instruments DM3730 SoC [90]; a Collins Aerospace CMU-900, an electronic system

Table 4.1: Jetset's primary targets and summary statistics regarding peripheral modeling.

	RPi2	**BBXM**	**CMU-900**	**SEL-751**
OS/SW	Linux 4.19.y	X-Loader	VRTX-32	G5.1.5.0
Peripheral inference				
Wall-clock time	6m43s	5m15s	5m20s	2h34m51s
Blocks in code base	238,792	872	55,016	141,750
Total blocks executed	81,194,393	20,198,824	53,143,508	3,351,484,857
Blocks executed on path	81,194,393	20,198,824	27,517,932	3,351,484,857
Unique blocks executed	43,157	484	776	11,364
Unique blocks executed on path	43,157	484	731	11,364
MMIO writes (ignored) on path	84,060	938	1,308	32,480
MMIO reads (symbolic) on path	83,857	3,633	242	704
MMIO write addresses on path	40	244	13	68
MMIO read addresses on path	37	61	5	26
Devices accessed	6	11	5	5
Peripheral synthesis				
Wall-clock time	3.16s	5.64s	0.018s	5.61s
Total Symbolic Variables	1,384	3,633	242	704
Total Constraints	5,226	8,353	756	11,142
Constraints per variable	3.78	2.30	3.12	15.83
Average trace length	37.4	59.56	48.4	27.08
Median trace length	1	3	5	2
Maximum trace length	1076	2,770	215	343
Emulator execution to goal				
Wall-clock time	8s	101ms	289ms	1m1s
Total blocks executed	81,454,594	20,198,656	27,519,080	3,351,502,947
Unique blocks executed	43,255	483	731	11,364
MMIO writes (ignored)	83,915	938	1,882	32,480
MMIO reads	83,857	3,633	242	704
MMIO write addresses	43	244	13	68
MMIO read addresses	27	61	5	26
Devices accessed	6	11	5	5

used on many Boeing 737 aircraft, responsible for handling digital communications between the aircraft and ground stations with an AMD Am486, Intel 486-compatible processor; and a Schweitzer Engineering Laboratories SEL-751 feeder protection relay, used to protect power grid systems, leveraging a MCF54455, a 32-bit microprocessor implementing the ColdFire ISA. The statistics on the emulation of these systems are given in Table 4.1.

Details on the emulated and symbolically executed versions of the execution are given by the top and bottom portions of the table. Differences, in generally, are explanable due to the backtracking of symbolic execution when it hits an infinite loop (in these cases Jetset must re-execute code and take a different path), and due to slower inference-stage execution. In the case of the Raspberry Pi, this led to an SD host controller command timeout, resulting in an error message and a register dump. During emulated execution with synthetic devices, the SD host controller initializes without a command timeout, thus, the executed blocks counts differ; however, the resulting emulation is resilient.

We also evaluated Jetset on the nine publicly-available real-world pieces of firmware used to evaluate P^{2}IM[23], and all of these rehostings were trivially successful. In general, the firmware images were non-complex and not symbol-stripped, making their analysis straightforward. The P^{2}IM used four different ARM SOCs, two Cortex M-3 (STM32F103RB and SAM3X8E), and two Cortex M-4 (STM32F429ZI and MK64FN1M0VLL12) CPUs. Five of these systems used Arduino as their operating system, one used the RIOT operating system, and three operated on bare metal.

4.3.1 Full Hardware System Emulation Fuzzing

The useful application of the emulations generated by Jetset required both the development of systems for *fuzzing* and *live binary rewriting* (in the case of the CMU. For this purpose, we reduced the requirment firmware modification imposed by the FirmAFL [26] QEMU-AFL combined fuzzer. A simplified depiction of our final architecture is given in Fig. 4.1.

We isolated each fuzzer instance into an system-resource isolated Docker [91] container, and then shared a file system folder between these instances. Docker was used to maintain isolation in the process tree and shared locks / backing files used by the QEMU emulation of the target systems. Within each fuzzer instance, file system folders were instantated to maintain the initial state of the emulated system, including backing files for flash memory and other peripheral devices, and an installation of QEMU and AFL with either a ground-truth implementation of the system emulation or a Jetset-generated

emulation. For speed, these backing files were mounted into a RAM filesystem, rather than disk, and each container was pinned to one of 32 CPUs. In order to provide the initial state for QEMU, a Checkpoint-and-Restore in Userspace (CRIU) snapshot was used [92]. Inside each container, the QEMU instance was run, reading inputs from AFL and writing outputs to the shared folder. QEMU was instrumented with hooks which would introspect the guest system for crashes or hangs and then signal to the host AFL instance within the container to reset the entire QEMU process tree from a snapshot. Because the process tree was reset rather than a single process, peripheral models and coprocessors running in separate child processes would also be reset, simplifying the maintenance of state.

The two targets of our dynamic analysis using this fuzzing framework were the Raspberry Pi and the CMU-900. Both fuzzing sessions targeted the OS system call boundaries of Linux and VRTX, respectively. While no novel vulnerabilities were found in Linux, all recovered fuzzing outputs were equivalent between the emulateed and physical versions of the Raspberry Pi. The CMU-900, however, had significantly more successful results. The AFL fuzzer found 2963 unique code paths during 200 hours of fuzzing, over 200 of which resulted in meaningful crashes. One of these code paths, crashing on a function return, was bootstrapped into a privilege escalation vulnerability using a ROP chain by the author. We next discuss the specific results of fuzzing each of these devices.

RPi Fuzzing Because the Linux kernel is used widely, we did not expect our testing to identify new vulnerabilities in its system call interface. Instead, our goal was to determine whether our synthesized configuration *behaves the same way:* for all three implementations, we monitored the response of the Linux kernel to each system call and recorded which of the following four observable behaviors resulted:

- **Kernel "oops" or panic.** Both indicate a kernel fault, pointing to a potentially exploitable bug. A kernel "oops" does not halt the system, while a panic does.
- **Process killed.** The kernel kills the process issuing the system call. In our configuration, the only process is the init process, leading to a kernel panic with a unique error message. Under normal circumstances process death would not result in a panic.
- **System call return.** The kernel returns to the calling process. We recover any set errno and return values.

In our experiment, we issued 1,571,576 distinct system calls from our init process stub to the Linux kernel running in QEMU with our synthetic devices, resulting in 123,198

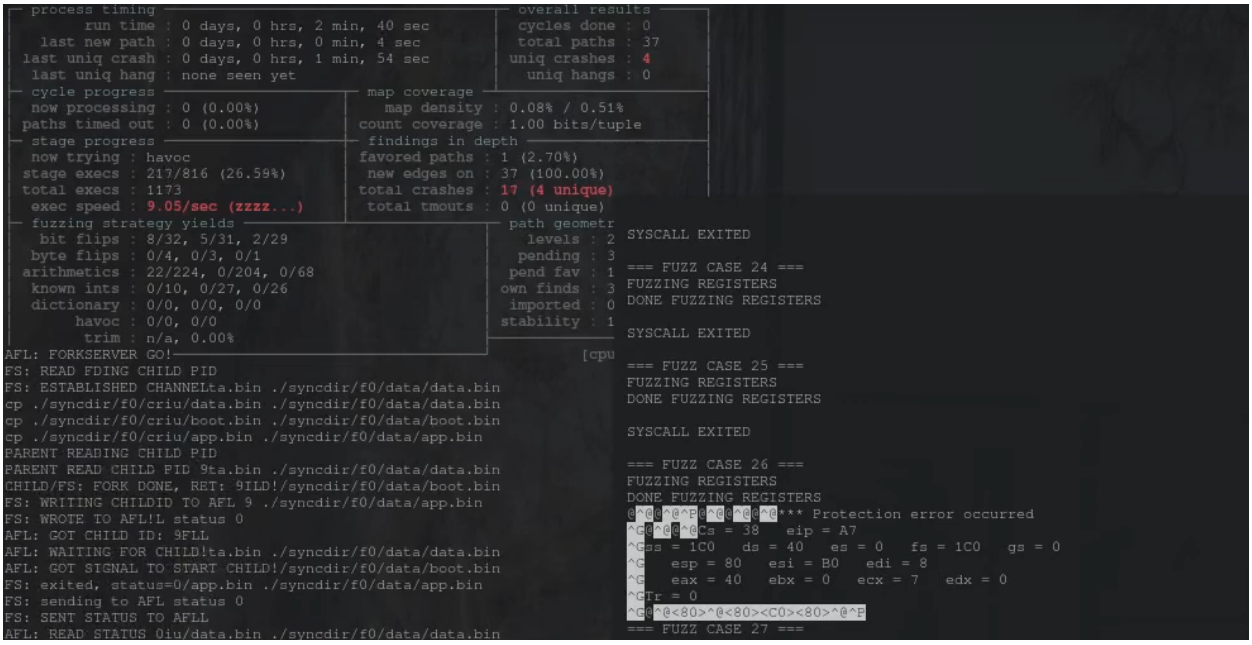


Figure 4.3: A screenshot of the Jetset fuzzing framework (left) discovering a crash (right) in the CMU-900.

unique codepaths. Of these, none resulted in a kernel "oops" or kernel panic. 51,638 resulted in the kernel killing the init process, and 71560 in the system call returning to our user process. We then carried out the same experiment (using the same exact system calls) using the official QEMU Raspberry Pi 2 configuration with manually-implemented devices. In all 123,198 cases, we observed the same behavior in both the synthetic and manually-implemented configurations, down to fuzzing paths discovered, error stack traces, `errno` values, and system call return values.

To compare our (synthetic) implementation against the real hardware, we selected a random sample of 14,661 test cases. We then booted the Raspberry Pi 2 kernel on the target board and used a custom init process to read system call parameters from a serial port, issue the system call, wait three seconds, and then reboot the system. Using this interface, we issued 14,661 system calls on the target hardware. If the system call returned, our init process printed `errno` and the return value to the serial console. Otherwise, we relied on kernel serial console output to determine whether the init process was killed or whether the kernel encountered a fault ("oops" or panic). We observed the same behavior on the physical hardware as in the emulator with synthetic devices.

CMU-900 Fuzzing The mainline QEMU distribution implements four of the peripheral devices used by the CMU-900 (the real-time clock, interrupt controller, interval timer, and

serial controller). We created a manual, ground-truth QEMU configuration mapping these devices at addresses expected by the code, which allowed us to compare the behavior of the emulated system with our synthetic devices against a QEMU configured with their full implementation. As with the Raspberry Pi 2, we compared the behavior of the two systems by issuing system calls from unprivileged task 1. A screenshot of the fuzzing framework is shown in Figure 4.3.

AFL found 2963 unique crash code paths during 200 hours of fuzzing. To compare the behavior of our two QEMU implementations (synthetic and manual), we compared the debugging console output produced by the CMU-900.[2] In the case of a successful system call return, the CMU-900 continues with its normal unprivileged task startup sequence. In the event of a protection violation, the CMU-900 prints a wealth of debugging information, which we use to determine whether the two configurations behaved similarly.

Of the 2963 execution paths discovered by fuzzing, 2884 (97.3%) code paths exhibited identical behavior. Another 36 (1.2%) had the same outcome, but differed in the values in some of the registers. The remaining 43 (1.5%) also had the same outcome, but differed more extensively in the output generated.

Manual analysis of the 2963 execution paths led to the discovery of a privilege escalation vulnerability. The vulnerability occurs because a single byte can be "leaked" from unprivileged code into the offset of a call instruction in the VRTX kernel. One of the 256 potential values for this byte results in the call targeting the middle of a function. From here, a Return-Oriented Program (ROP) chain can be used to modify the global descriptor table (GDT). A few instructions later, the GDT modification causes the kernel protection error handler to fire. However, the GDT modification changes the base address of the data and stack segment used by the handler so they overlap with an unprivileged data segment. The handler includes a far call whose address is dependent upon a read from the corrupted data segment. A malicious address can be given to this far call, leading to a second ROP chain which further modifies the GDT. This ROP chain changes the address range limits for privileged code and transfers control to unprivileged, writable memory while remaining in processor ring 0.

We were able to validate the exploit on another CMU-900 (one with a slightly different part number and memory layout) after some minimal adaption and a live binary rewriting process paired with a local code execution exploit which "poisoned" the VRTX operating system with shim code. Due to small changes in the VRTX kernel between device versions,

[2]We would prefer to compare the synthetic device QEMU instance to the actual hardware, however, did not want to risk making the device inoperable, one explicit benefit of the emulated copies.

we discovered the exploit's control transfer required supplying one ROP gadget address via a segment rather than a data register and changing some gadget offsets.

4.3.2 Leveraging Emulation in Avionics Exploit Development

While the loading capability used to introduce software into the physical CMU was provided by the Triton Avionics Testbed [93], significant additional live binary instrumentation was needed to evaluate the exploit discovered above, and the techniques that were developed in this process allowed us to develop a mechanism for poisoning the CMU-900's VRTX operating system and led us to the discovery of three sets of messages capable of remotely disabling the CMU-900. Because Aircraft Communication and Reporting System messages also have a "broadcast" mode, these messages could be used to crash a large number of CMU-900 devices simultaneously. Due to their sensitivity, we omit the details of these messages from this dissertation and focus on techniques for their discovery, noting only that one message generates a buffer underflow where memory is allocated beyond the appropriate length and the other two cause excessive memory allocations which drain the availability of new heap memory.

The ability to run the CMU-900's VRTX operating system in an emulated environment was critical to the development and testing of these techniques and associated exploits. The emulated copies of the CMU-900 allowed us to attempt numerous instrumentation approaches and gain visibility into implementation flaws, a task which would have been significantly more difficult on the physical device.

Bootloader and OS Initialization Replication Once our malicious software is loaded into the CMU-900 using the dataloader, control is effectively transferred entirely to the malicious code as a bootloader, negating any prior operating system and hardware device initialization. It was thus necessary to replicate the bootloader and operating system initialization process in the emulator to ensure that the malicious code would be able to run in the same environment as it would on the physical device. So, in this initial case, the malicious code would first relocate itself to an unused portion of RAM, transfer control to this new location, replicate the bootloader and initialization behavior necessary to reset the system back to an appropriate initial state, and then return control to to the orginal VRTX operating system.

Developing this replicated functionality required significant manual extraction and analysis of code using Ghidra. The complexity of dealing with self-relocating code and the specific magic numbers involved for resource initialization led this "poisoned" version

of the VRTX boot sequence required hundreds of lines of hand-written x86 assembly. The development of this *in situ* emulation also required significant universal quantification over possible relocation addresses as an alternative to the manual process of identifying a suitable relocation address, a unique benefit of using an emulated system, and served as one inspiration from the redaction attacks presented in Chapter 5.

Flash Code Relocation and Shim Injection For the purposes of analyzing the system for crashing inputs and testing the system call paths discovered by fuzzing in the previous section, it was not enough to *replicate* the boot process. Thus, we also performed binary rewriting of operating system code in order to inject instrumentation routines for the purposes of dynamic analysis.

This was complicated by the fact the CMU-900 executed a significant portion of its code out of flash memory, which was considered to be read-only in order to avoid damaging the physical unit. It was therefore necessary to relocate the portions of code that were to be instrumented into unused, writable RAM memory, and update associated pointers. This was a two stage process, first consisting of corrupting the interrupt vector table in order analyze portions of the code that were in use at runtime for critical functions and relevant to exploitation, and second, updating non-relative memory addressing data to point to the relocated code segments.

However, once the appropriate relocation operations were complete, it was possible to inject additional shim code at arbitrary locations in system memore using binary rewriting tactics. The primary rewriting tactic used was the augmentation of x86 call locations. The precise method used is detailed in Section 3.2 of this dissertation, and code is given in Figure 4.4 With shim instrumentation in place, it was trivial to manually identify potentially crashing inputs by examining data flows within the system.

4.4 OPEN CHALLENGES IN REHOSTING

Returning to the primary subject of Jetset, the approach of targeted rehosting works well for firmware running on a variety of embedded system architectures across multiple application domains. However, this approach to peripheral modeling and system emulation is not without limitations, which we break down into *internal* and *external* semantic challenges.

```
/* eax stores the original offset in the code. we read from
   this offset to get the relative offset the call lands at.
   We then change that offset so it is relative to the
   location of the trampoline call instruction. We then set it
   into the trampoline call instruction. */
"mov    $" TRAMPOLINE_LOC ",%ebx\n"
"add    $3,%eax\n"
"mov    %eax,1(%ebx)\n"
"sub    $3,%eax\n"
"mov    %eax,%ebx\n"
"add    $0x" RELOCATE_ACARS_CODE_BASE "00120,%eax\n"
"mov    (%eax),%eax\n"
"shl    $0x8,%eax\n"
"sar    $0x8,%eax\n"
/* Add three bytes to account for the three storage bytes */
"add    $0x3,%ebx\n"
/* Add relative base of call so it is relative to seg base */
"add    %ebx,%eax\n"
"sub    $" TRAMPOLINE_LOC_OFF ",%eax\n"
/* Subtract 10 bytes relative to the tramp call return  */
"sub    $0xA,%eax\n"
/* Write the jmp instruction of the trampoline loc */
"mov    $" TRAMPOLINE_LOC ",%ebx\n"
"mov    %eax,6(%ebx)\n"
```

Figure 4.4: Code for live binary rewriting of call instruction in the CMU-900 firmware.

Internal Semantics The paths explored by Jetset through firmware are not necessarily ones that may ever be returned by the hardware in the original system; however, the path taken is one that is acceptable to the firmware—no interaction with any of the peripherals results in a boot failure. While the execution of firmware running on physical hardware is constrained in its behavior by how the physical peripherals really behave, these system constraints are external to the firmware and cannot be inferred without auxiliary information about its behavior. To effectively emulate a system, it is not necessary not only to explore a single execution path, but to reveal the connections and meaning implied by multiple execution paths, a topic we will approach in Chapter 5 and refine in Chapter 6.

External Semantics Jetset does *targeted rehosting* in that it only constructs an emulator that is sufficient to emulate the software component-under-test. However, rehosting may have no method of deducing the semantics of devices that the system depends upon if the only information it considers is that which is contained within the system specification which remains after these devices are removed, e.g. firmware. Jetset could be extended to synthesize more complex, stateful peripheral models as well as identify known peripherals with existing emulator implementations using additional information from external sources.

Included in external semantics is the information requires to solve the dynamic analysis challenges presented by Section 4.3.2, which exist outside of the simple execution of a given system model. While this information is of less general application, it still plays a critical role in quantifying the security of the emulated system, as evidenced by its utility in exploit development.

4.5 SUMMARY

In this chapter, we discussed one approach to the emulation of complex systems, targeted firmware rehosting. The benefit of this approach is its ability to synthesize peripheral hardware devices for the analysis of real-world embedded systems. Particularly, it considers the available information about the system to derive models of unknown components. In the next chapter, we will develop an alternative approach which addresses some of the shortcomings of looking at a strict derivation from available information for resolving unknown components: universal quantification over possible missing information, and see how this approach applies to the security of real-world redactions.

Chapter 5: Digital Forensics Using Simulations

In this chapter, we explore the application of sysem simulation for the purposes of digital forensics, particularly in the analysis of PDF documents. We begin by introducing facets of the PDF format and redaction software that are relevant to the analysis of redacted documents. Then we move on to discuss the extraction of models of PDF producing software, and the use of universally quantified inputs in determining the text that is removed during redaction. We then introduce the most impactful discovery covered in this dissertation: the fact one to two word *exising* redactions, where the text is removed from the page, do not actually work. Finally, although it is not immediately related to emulation, we see it necessary to discuss the ethical implications of these findings as they present a significant threat to individual privacy.

5.1 INTRODUCTION

As noted in the last chapter, the one core element of emulation is the modeling of unknown components of the system, and one approach to this problem is *universal quantification*, which tries every possible match for the missing information.

The representation of missing information is no better encapsulated than by the centuries-old process of redaction, which removes or obscuring parts of a document to prevent its disclosure. In the past this was performed by black marker or by physically cutting parts of the document out with scissors. However, with the move to digital, paperless representations of documents, the process of redaction is typically performed by software tools. The process of removing information from digital documents is error-prone, as demonstrated by several high-profile examples [94]. However, while past work has considered the possibility of supplying all inputs to a system in a *brute-force* attack, no existing work has considered the role the PDF file representation plays in redaction security, and redaction's vulnerability to such guess and check attacks.

In this chapter, we begin by noting that the width of the characters in a proportional-width font can leak information about redacted text, and extend this notion of leaked information to the PDF file's representation of sub-pixel-sized chatracter positioning shifts. These shifts can be used to decude far more information about redacted text than width alone, and failing to understand these shifts exist leads to incorrect or imprecise deredaction, serving as a useful parable for the more general field of modeling systems.

We were the first to identify this problem of sub-pixel shift information, leading us

to measure the severity of information leaks in PDF text redactions. We discovered methods for breaking redactions occurring on documents processed by optical character recognition (OCR) software and find that *rasterizing* a document (converting the PDF to an image) increases the security of redactions but does not remove information leaks. We also find a typical PDF document authored in Microsoft Word leaks about 13 bits of information about a redacted surname. This is enough for an attacker to identify a single individual from 8,000 candidates (say, employees at a company or potential confidential informants in a gang), or reduce the 3.9 million possible first initial, surname pairs from US census and Social Security Administration data to a set of only 200 potential matches for a redacted name.

Although it did not relate to our core theme of universal quantification, we also considered *nonexcising* redactions which do not remove any text from the document, and leave it directly encoded in the original PDF file. The text of these redactions can be copied and pasted from the source document. Our study and techniques located thousands of nonexcising redactions in US court documents, resulting in our notification of both the US court system and the Free Law Project about these redactions.

To understand the methods by which systems leak redacted information, we examined 11 popular PDF redaction tools and find that two do not work at all.[1] These two tools leave the text underneath the selection marked for redaction in the PDF document. The remaining tools, including those from Adobe Systems, remove the redacted text and replace it with a black rectangle. Without additional defenses, this practice leaks significant glyph shifting information which can be used to deduce redacted text.

We built a tool Edact-Ray, which *simulates* the operation of common PDF production software, such as Microsoft Word, and contains analysis subsystems which can be used to identify, break, and fix redaction information leaks across millions of PDF files. Using dictionary attacks on these simulations, Edact-Ray allows an attacker to test which strings from a dictionary of candidates create the same PDF output as the original redacted PDF file. We applied Edact-Ray to study the impact of our findings on three publicly-available corpora of PDF documents, breaking redactions in several documents of historical importance.

The resulting findings present a clear invasion of privacy for people whose names have been redacted and may be used to circumvent redactions used in censorship. Thus, we have notified more than 22 organizations whose redacted documents were affected, as well as vendors of leaky PDF production and broken redaction tools, and continue to

[1] As these findings are not entirely relevant to the subject of this dissertation beyond the minor application of dynamic analysis, we refer readers to the original paper for more details on redaction tools [10].

work with several of these groups to remediate the problem.

In the following sections, we provide the reader with the necessary background to understand how these redactions are broken through simulation and modeling of glyph positioning algorithms.

5.2 BACKGROUND ON PDF TEXT REPRESENTATION

The security of PDF text redaction depends on the specification of the PDF document. We consider two types of PDF documents. One is a raster image of the original document. The other PDF document type contains text data for both the font and the layout of each character (*glyph*) on the page. We focus our discussion on non-raster PDF documents, however, the concepts presented apply to raster PDF documents.

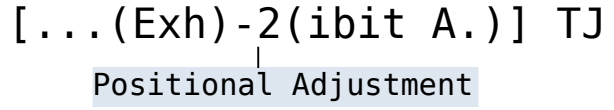


Figure 5.1: The TJ text showing operator, which specifies the glyphs to render and, by reference to a font object (not shown), their widths.

PDF documents can render text in innumerable ways (e.g. by embedding an image in the document), though the most common is through use of a text showing operator, one of which (TJ) is depicted in Figure 5.1. The TJ operator takes as arguments a string of text and a vector of *positional adjustments* which displace the character with respect to a default position. This default position is usually a fixed offset from the previous character equivalent to the *advance width* of the previous character defined elsewhere in the PDF document.

For our analysis, we converted the complex set of PDF text rendering operations into a uniform *intermediate representation*, consisting of a minimal set of metrics (e.g. font size) and a series of TJ operators. The intermediate representation presents PDF text as, effectively, a set of advance widths and *glyph shifts* which are the sum of all the individual positioning operations applied to a glyph. This conversion was necessary to account the large number of ways in which text can be rendered in PDF documents.[2]

The positional adjustment in Figure 5.1 is -2 *text space units* between the *h* and *i* glyphs.

[2]For example, in the case of a TJ operator, the actual *glyph shift* includes any offset due to a positional adjustment as *part* of the calculation of its value.

Text space units express glyph shifts, where 1,000 units almost always[3] equals the point size of the font times 1/72 of an inch. For a 12-point font, 1 unit equals 1/6,000 of an inch (0.0042 mm).

Glyph advance widths and glyph shifts create a security concern, as an attacker can supply all possible *inputs* to a simulation or high-level emulation of the tool used to create the document and create two significant security risks:

- The precise width of the redaction can be used to eliminate potential redacted texts.
- Any non-redacted glyph shifts conditioned on redacted glyphs can be used to eliminate potential redacted texts.

Examples of these output glyph width and shift descriptions as given by the PDF format are provided in Table 5.1. In this table, "Edge/Firefox, Print/PDFViewer, Save" can be interpreted as "using the edge or firefox browsers' PDF viewer or print dialog, hit save" and results in the right column's displacements. We do not include strings of numbers in our analysis as digits' glyph advance widths and effect on glyph shifts is almost always identical, making them effectively monospace, even if typeset in a variable-width font. Some uncommon fonts may have non-identical digit glyph advance widths.

5.2.1 Glyph Shifts

The width of a PDF redaction depends on glyph shifts. Without accounting for glyph shifts, redacted text guesses are imprecise and must account for error, reducing the potential of finding a unique match for redacted content. The glyph shifts present in a PDF document are dependent on the specific *workflow* used to produce the PDF document. This includes an originating software, called the *PDF producer* by the ISO 32000 PDF standard [95], and any software that may modify the PDF file contents thereafter, including, for example, a redaction tool. A given workflow creates a specific pattern of glyph shifts, determining, in part, the security of any redacted text. We identify two types of glyph shifting schemes:

- *Independent*: the glyph shifts for a given character are not dependent on any other character in the document in any way.
- *Dependent*: the glyph shifts for a given character are dependent on some other character in the document in some way.

[3] PDF offers the ability to redefine the text space and Edact-Ray accounts for this.

Table 5.1: Several possible PDF workflows. The left column indicates the operations which to produce the representation on the right.

Software Stack, Workflow	Description of Positioning Scheme
Word 365, Windows 10	
Edge/Firefox, Save	[(E)7(xhi)7(bi)7(t)7(A)-6(.)]
Edge/Firefox, Edited, Save	[(E)7(xhi)7(bi)7(t)7( )-1.75(A.)]
Edge/Firefox, Print/PDFViewer, Save	[(E)-13(xhi)28(bi)28(t)-34(A)-27(.)]
Edge/Firefox, Edited, Print/PDFViewer, Save	[(E)-13(xhi)28(bi)28(t)-34( )-3.17(A.)]
Edge, Edited, Print/PDFViewer, Print	Stream of Vector Graphics Commands
Firefox, Print, Save	Stream of Vector Graphics Commands
Firefox, Print, Print	[(Exhi)7(bi)7(t)7(A)-8(.)-20( )]
Chrome	Subset of Edge Outputs
Desktop, Save (all flows, including loaded again in Adobe)	[(Exhibi)-2(t A.)]
Desktop, Edited Save (all flows)	[(Exhibi)-2(t)-2.67(A.)-2( )]
Desktop, Edited/Unedited Print	[(E)-0.8398(xhi)-0.8320(bi)-0.8320(t)-0.8320( )-10(A)-0.1680(.)10( )]
Word 365, Mac OS	
Safari, Chrome, Firefox, Edge	Subset of Windows 10, Edge
Desktop, Save (all flows)	[(E)0.2(xhi)0.2(bi)0.2(t)0.2(A)-0.2(.)-0.104( )]
Desktop, Edited Save (all flows)	[(E)0.2(xhi)0.2(bi)0.2(t)0.2( )-0.136(A.)0.232( )]
Word 2016, Windows 10	
Save	[(Ex)-8(hibi)6(t A.)]
Save, Edited	[(Ex)-8(hibi)6(t)-2.67(A.)-2( )]
Print	[(E)-0.8398(x)-10(hi)-0.8320(bi)7(t)-0.8320( )-10(A)-0.1680(.)10( )]
Google Docs, Windows 10	
Save (All flows)	[(Exhibit A.)]
Save, Adobe, Save	[(Exhibit)55.838(A.)], [(AAA)128.417(VVV)]
Chrome/Edge, Print, Adobe PDF Save	[(0123435ÿÿÿ78ÿÿÿ)] (Actual embedded text is clobbered.)
Edge, Print, Print	Stream of Vector Graphics Commands
Firefox, Download/Print, Firefox Viewer, Print, Print	Stream of Vector Graphics Commands
Google Docs, Ubuntu 21.04	
Firefox/Chrome, Save (all flows)	[(Exhibit A.)]
Firefox, Download, Firefox Viewer, Print	File rasterized
Chrome, Print/System Print	[(Exhibit A.)] (Spaces tripled)
Google Docs, Mac OS	
Chrome/Firefox/Safari, Download/Print, Save	[(Exhibit A.)]
Chrome, Print, Save as Adobe PDF	Same as Ubuntu, Chrome, System Print
Chrome, Print, Preview/Sysdiag Print, Export	[(E)0.2(hi)0.2(bi)0.2(A.)], (Spaces Tripled)
Firefox, Print, Preview, Export	[(E)0.2(hi)0.2(bi)0.2(A.)]
Firefox, Download, Firefox Viewer, Print	File rasterized
Quartz PDFContext (Apple Pages), Mac OS	
Export/Print/Preview, Save	[(E)0.2(xhi)0.2(bi)0.2(t)0.2( )55.2(A)-0.2(.)], [(AAA)129.0(VVV)]
Print, Save-as-Adobe-PDF	[(Exhibit A.)], [(AAA)128.8(VVV)]
Word 2012, Windows 10	Same as Word 2016, Windows 10
All Word Versions, Windows 8.1 and Older	Subset of Windows 10
No Disp. 600 DPI, Adobe OCR	
OCR Only, Save	[(E)-2.3(x)-2.3(h)-2.3(i)-2.3(b)-2.3(i)-2.3(t)-2.3( )*[79.75 Td - 2.3 Tc]*(A)3.3(.)3.3( )3.3()]
Edit or OCR, Edit, Save	[(E)-0.12(x)-0.12(h)-0.12(i)-0.12(b)-0.12(i)-0.12(t)-0.12( )*[71.6 Td - 0.12 Tc]*(A.)]

Table 5.2: Glyph width equivalence classes for the specific default version of Times New Roman used by Microsoft Word.

569	ijlt	1251	ELTZ
683	Ifr	1366	BCR
797	Js	1479	ADGHKNOQUVXYw
909	acez	1593	m
1024	bdghknopquvxy	1821	M
1139	FPS	1933	W

These correspond to the notions of dependent and independent information in the recovery of models of systems introduced in Chapter 2. We call independent schemes *unadjusted* when there are no shifts on any character. Google Docs' *Export to PDF* option produces an unadjusted scheme.

Equivalence Classes.

Before discussing these schemes further, we introduce the idea of width and shift equivalence classes. A shift equivalence class is a set of lists of glyphs of the same length with identical shift values. A width equivalence class is a set of glyphs and associated shifts with the same width.

Table 5.2 gives an example of the width equivalence classes for glyphs in a Times New Roman font[4] without shifts, next to their widths as specified by the font file. Each number is the width in given to the glyph by the font file and each set of letters has equivalent widths. The glyphs *I*, *f*, and *r* are exactly half the width of the glyphs of *B*, *C*, and *R*. When typeset using Times New Roman, the words *martian*, *templar*, and *mineral* all have the same width, as do the anagrams of those words *tamarin*, *trample*, and *railmen*.

The PDF specification does not include any specific signifiers for redacted text. However, residual specification information after redaction, such as glyph positions, can be used to reasonably rule out large numbers of candidate width and shift equivalence classes for redacted text. None of the prior words in this paragraph are in the width equivalence class of the word *cat*.

[4]Different versions of a font can exist. We use the default versions available from Microsoft throughout this paper.

Independent Schemes.

In an independent glyph shifting scheme, the security of a redaction may be considered dependent on the size of the width equivalence class indicated by the PDF document's residual glyph positioning information. That is, the positions of glyphs prior to and succeeding the redaction may leak the width of redacted text. The scheme's specific glyph shifts can make a given width equivalence class leak more or less redacted information by making width of individual glyphs more or less unique.

As an example, consider redacting a single letter *l* as opposed to *m* in the Times New Roman (TNR) scheme from Table 5.2. *m* is the only glyph in its width equivalence class, so it may be possible to determine the letter *m* was redacted uniquely. Whereas if *l* is redacted, then the residual information indicates the redacted letter could be any one of *i*, *j*, *l*, or *t*. However, if *l* were always accompanied by a glyph shift distinguishing it from these other three letters, then the scheme would leak more information.

We acknowledge that there is no *guaranteed* correlation between glyph positions before and after redaction. Redaction may reposition glyphs in such a way as to destroy accurate width information. However, we found this is almost never the case for commonly accessible redaction tools [10]. We were also informed that in some contexts there are (legal) restrictions on changing the glyphs or glyph positioning of a redacted document.

Scanned Documents. Many documents with independent shifting schemes are the result of an optical character recognition (OCR) process. This process embeds a non-raster representation of the document's text in the resulting PDF document. This often allows the document's text to be searched and copied from by software tools.

Non-raster deredaction attacks work on documents that are scanned, OCRed and then redacted, not on documents that are redacted, scanned, and then OCRed. We note that attacking the OCR overlay is not always as straightforward as attacking a PDF produced without OCR, and requires a separate model or simulation.

Dependent Schemes.

A dependent scheme is more dangerous to the security of redacted text than an independent scheme. In these schemes non-redacted glyph shifts can be dependent upon redacted glyph information, because the non-redacted glyph shifts can be determined *before* redaction.

Microsoft Word "Save as PDF". In this chapter, our primary simulations model a class of dependent schemes defined by the Microsoft Word software's *Save As PDF* command. We

```
1  for (int j = i + 1; j < vs->size(); j++) {
2      t = ttfScaledWidths[j] / 1000;
3      d = internalMSWordWidths[j] / internalMSWordFontSize;
4      ttf += t;
5      msWord += d;
6      disp = ttf - msWord;
7      if (
8          ((disp > 0.003) || (disp < -0.003)) &&
9          i != vs->size() - 1
10     ) {
11       int adj = disp * 1000 + 0.5;
12       vs->setShift(j, adj);
13       ttf = msWord = 0;
14     } else {
15       vs->setShift(j, 0);
16     }
17 }
```

Figure 5.2: Snippet of reverse engineered code representing how Microsoft Word leaks redacted character information into non-redacted characters in a PDF document.

reverse-engineered the glyph shifting scheme produced by this command in Microsoft Word for Windows desktop versions 2007 to 2019. This process took around several months. Word 2007 to 2016 use one scheme, and Word 2019 to present versions use another. These two schemes affect thousands of real world document redactions (Sec. 5.4).

The studied dependent schemes accumulate a What You See Is What You Get (WYSIWYG) error measurement for each glyph from left to right across each line of text. If redacted content is not removed from a Word document before running "Save as PDF", redacted glyph positioning information affects the accumulated error value. *Thus information about the content of a redaction is leaked into the shifts applied to non-redacted characters.*[5]

Figure 5.2 depicts this behavior. Word's internal representation of the layout of characters for display purposes does not exactly match that of the TrueType Font (TTF) embedded in the PDF document. Word corrects for this small error between the two formats through use of glyph shifts. Surprisingly, these modifications are done independent of the user's screen resolution.

We note the internal widths used on line 3 of Figure 5.2 are determined by a loop with no overflow reset and the redacted information held by the accumulator *is not zero*

[5]Different text justifications can leak *more* information.

after a single shift is written. We refer the reader intereseted in the specific details of this algorithm to our original paper [10].

Word's shifting scheme depends on the document's edit history. Changing a character inside of a Microsoft Word document splits the internal representation of the text fragment containing the character into two fragments. This fragmentation resets the accumulation of glyph width error and affects the shifts emitted for a line of text. Edact-Ray accounts for this by simulating all potential edits to redacted text.[6]

We validated our Word model on hundreds of lines of Wikipedia text rendered to PDF by Word, several manually-crafted test cases, and text fragments from redacted documents found in the wild.

Locating Redactions Briefly, to evaluate our forensic capabilities, we also developed a redaction location algorithm for PDF documents, based upon an implicit model of how redaction often appear as outputs of PDF document redaction systems.

For nonexcising redactions, we first detected filled boxes drawn over text. This creates a large number of false positives as it does not consider whether the rectangles are actually covering text in a meaningful way, so we also check whether the pixels in the bounding box of each PDF glyph are all the same color. This does not handle complex cases (e.g. using an image to redact text), but we did not find cases of these more complex styles of redaction in practice.

We manually validated this nonexcising redaction location algorithm on PDF documents from RECAP, a website hosting US court documents, and discovered a false positive rate of 4%. An additional 7% of these documents properly changed the underlying text (e.g. to REDACTED) or contained unintended redactions (e.g. a black box covering non-sensitive text). This resulted in 710 documents with nonexcising redactions, many of which included entire paragraphs of text. The algorithm encounters false negatives when it cannot identify rectangular draw commands. We did not encounter false negatives: redaction draw commands are rarely ambiguous enough to prevent detection.

For excising redactions, we first identified spaces larger than a single space character between each pair of words in a document. It then analyzes the pixel color values between the words to identify a drawn rectangle.[7] We evaluated our excising redaction location algorithm's accuracy on a random sample of 1,000 redacted pages from real-world documents, and compared against Lee's [96] prior work on redaction location and against

[6] We found edits have no significant impact on the efficacy of deredaction, as the information leaked is *at least* as much as an independent glyph shifting scheme.

[7] We did not attempt to locate redactions without some visually signifying feature.

manual identification. Lee's method flags every black rectangle drawn as a redaction and has a high false positive rate for some classes of documents (around 30% for FOIA and nearly 100% for RECAP). With respect to false positives and false negatives, our algorithm is equivalent to manual analysis when locating what we deem to be vulnerable redactions.

5.3 MODELING AND SIMULATING GLYPH SHIFTING

For a given PDF generation and glyph positioning workflow, it is possible to construct a precise simulation or model of the workflow's operation and layout of glyphs. In this chapter, the process was peformed using WinDBG's data flow analysis [74] and a series of carefully crafted scripts for extracting information on how glyph positions are decided.

Extraction of System Simulations To extract simulations of glyph shifting schemes for Microsoft Word, Adobe Acrobat, and other systems, we began by recording an execution trace of the production of a PDF document. In this document, we included easily identifiable text fragments, such as "Lorem ipsum" and "redacted", which excercised the glyph shifting scheme. Because a given software workflow can split its operations across multiple threads, we performed this recording in a virtual machine emulation running on a single core. Then, we scripted WinDBG to identify the last use of a `rep movs` operation with a pointer to the indentifiable string.

This resulted in roughly five different identifiable code paths under which the specific glyph shifts were decided. Code paths were discovered by using *time-travel debugging* to trace the operation which last operated on a data value using a script. The binary code for these paths was then extracted, decompiled, and then translated to a C-code representation that could be compiled as a standalone binary. Beyond the strict operational semantics of the system, it was also necessary to extract certain dictionaries from the binary, such as the internal width map used by Microsoft Word in order to determine the coordinate space "error" between Word's internal representation and the output PDF representation. We provide an example of a script which extracted such a dictionary in Fig. 5.3. This script operates on an execution trace recording of the production of a PDF with a separate glyph rendered on each line, e.g. "a", "b", "c". In addition to the above, a final manual analysis step was required to identify the *types* used in certain operations, e.g. float vs. double, as small floating point errors would have dramatic effects on output shifting information.

```
function handleInternalWidths() {
    let Regs = host.currentThread.Registers.User;
    let iWi = host.parseInt64(Regs.eax);
    /* hack to identify the newline character */
    if (iWi < 0x500) {
        let internalWidth = iWi.toString(10);
        internalWidths[characters[pos]] = internalWidth;
        pos += 1;
        if (pos == characters.length) {
            internalWidthsArr.push(toJson(internalWidths));
            internalWidths = {};
            pos = 0;
        }
        lineend = !lineend;
    }
    return false;
}
function invokeBP(bpCmds) {
    let Control = host.namespace.Debugger.Utility.Control;
    Control.ExecuteCommand('bc *');
    host.diagnostics.debugLog("Rewinding Execution...\\n");
    Control.ExecuteCommand('g-');
    host.diagnostics.debugLog("Done\\n");
    for (var i = 0; i < bpCmds.length; i++) {
        Control.ExecuteCommand(bpCmds[i]);
    }
    host.diagnostics.debugLog("Invoking BP...\\n");
    Control.ExecuteCommand('g');
    host.diagnostics.debugLog("Done\\n");
}
function invokeScript() {
    invokeBP(['bp /w
       "@\$scriptContents.handleInternalWidths()"
       0x65567138']);
    for (var i = 0; i < internalWidthsArr.length; i++) {
       host.diagnostics.debugLog(internalWidthsArr[i]);
    }
}
```

Figure 5.3: Example script for extracting the internal width map from Microsoft Word.

Universal Quantification of Inputs With C-code simulations extracted from respective PDF document production and redaction systems, it was then possible, for a given line of redacted text, to universally quantify all possibilities for the redaction's content. This dictionary attack aims to reproduce exactly the original behavior of the system when the document under analysis was produced: Edact-Ray measures the security of a given redaction by developing an information fingerprint (i.e. a hash of the width and shift equivalence classes) of all leaked glyph positioning information from the simulation. For each attempted guess at the redacted content, Edact-Ray matches the information fingerprint for this guess to the fingerprint of the same information recoverable from the source document.

If the guess information fingerprint does not match, that guess is ruled out as a potential redacted content. Thus, Edact-Ray's results are only as good as the dictionary which it quantifies over, and if the redacted text is not in the dictionary, Edact-Ray may return incorrect or no results, thus it is important to understand that in the determination of unknown facets of a system, it is impossible to *discover* the ground truth unless it is also possible to guarantee the dictionary used to guess and check is sound. Thus, while the generation of potential dictionaries could be automated using machine learning, we chose to perform this step manually in order to exclude a potential source of error. We also used dictionaries consisting of multiple variations of possible redacted terms in order to minimize the chances of not including the correct redacted content in our guess dictionary, and performed extensive validation of our results wherever possible.

Running on an eight-core consumer laptop (a ThinkPad T420) and simulating the Microsoft Word shifting scheme discussed in the previous section, Edact-Ray performs 80,000 guesses per second.

5.4 BREAKING REDACTIONS

Given the Edact-Ray tool, it was now possible to measure the security of redactions in a variety of real-world PDFs and synthetic cases. First, we were able to quantify the exact amount of information each shifting scheme leaked using a set of *synthetic* redactions, where we attempted redaction on text with known ground truth. For this we used text from the New York Times Annotated Corpus [97]. Then, we explored whether redactions in real-world documents could fall victim to these same simulation-based attacks, and quantified the number of broken redactions across thousands of PDFs.

Dictionaries The amount of information leaked also depends on prior information about the redacted text. For example, if we know the redacted text is one of 151,671 American surnames, then this redaction leaks at most $\log_2 151,671 \approx 17.2$ bits. In our experiment, we model this prior information as a *dictionary*, a set of strings from which we assume the redacted text is drawn.

Based on an examination of real redactions, we constructed 10 dictionaries.

- *Str.* All strings of 3–16 characters in length starting with a uppercase or lowercase letter followed by lowercase letters.
- *Acrn.* All strings of 2–5 uppercase characters.
- *Word.* English words including some proper nouns.
- *Ctry.* Official and common names of countries.
- *Rgn.* Names of regions, a superset of *Ctry*.
- *Natl.* Nationalities, demonyms, and adjectives of regions and nationalities, sourced from lists on Wikipedia.
- *FN.* American given (first) names.
- *LN.* American surnames (last names).
- *FI×LN.* All combinations of a name initial followed by surname (*LN*).
- *FN×LN.* All combinations of a given name (*FN*) followed by a surname (*LN*).
- *FNLN and FILN. FN×LN* and *FI×LN* filtered to only include combinations of name and surname that appear in the voter registration databases of the three US states. North Carolina [98], Ohio [99], and Washington [100] were chosen based upon the availability of publicly accessible data.

Table 5.3 lists the dictionaries and their statistics. *NYT Occ.* gives the number of occurrences of words from the given dictionary in the NYT corpus. $H_u(X)$ is the uniformly distributed information-theoretic entropy of the dictionary, given by $\log_2$ *Size*. For individuals' names, $H_e(X)$ is the entropy of the empirical distribution of dictionary words in the voter registration databases, and for all others it is according to the NYT corpus.

We use the voter registration database frequency for individuals' names to avoid bias present when using the NYT corpus to estimate the frequency of a given name. For example, "Al Gore" is more frequent in the NYT corpus than the population. Using the NYT for names would skew the results in our favor. We opted to provide a general representation of deredaction's efficacy where nothing is known about the name in question beyond population statistics.

The voter registration databases contained 1.7×10^7 names total. Note that if this measure was used instead of *NYT Occ.*, FN, LN, FILN, and FNLN would be identical to the population size.

Table 5.3: Dictionaries containing candidate texts used for evaluating deredaction. Stop words are excluded from the statistics.

Dict.	*Size*	$H_u(X)$	*NYT Occ.*	$H_e(X)$
Str	9×10^{22}	76.3	791,209,093	11.8
Acrn	1.2×10^{7}	23.6	4,674,379	10.0
Word	63,054	15.9	432,896,070	12.3
Ctry	566	9.1	3,905,371	5.9
Natl	509	9.0	4,081,019	5.4
Rgn	2.8×10^{6}	21.4	33,636,150	10.6
FN	100,364	16.6	71,031,188	10.3
LN	151,671	17.2	97,551,697	13.1
FILN	1.6×10^{6}	20.6	1,265,265	17.0
FI×*LN*	3.9×10^{6}	21.9	2,139,713	17.0
FNLN	8.9×10^{6}	23.1	3,650,063	19.6
FN×*LN*	1.5×10^{10}	33.8	14,440,238	19.6

Columns $H_u(X)$ and $H_e(X)$ are an *upper bound* on the amount of information a document can leak about a redacted element of that dictionary. If the amount of information leaked is equivalent to these numbers, then every word in the dictionary can be uniquely identified by leaked information when redacted.

Compared to the brute force approach of using *all possible* name combinations, using voter registration databases makes deredaction highly precise. However, we do not use *FNLN* and *FILN* in our evaluation of real-world documents because we do not want to bias our results by assuming what state the person (whose name was redacted) votes in or whether they are registered to vote.

Synthetic Evaluation We simulate redaction in a PDF created with unadjusted, Word 2007 and Word 2019 "Save as PDF" shifting schemes, in 10 point Times New Roman, Calibri, and Arial, the three most common fonts in our real-world document corpora, discussed next. These fonts account for 71.6% of lines in the 40,000 documents we studied. We include Courier as an example of a monospaced font. The simulated PDF, formatted for US Letter size paper, is left-justified with 1-inch margins, giving a 468 point line width.

We use a 10 point font size for our experiments as an upper bound on the amount of leaked information. Our original publication also includes results for a 12 point font size [10]. Because each 12 point line has fewer characters, the larger font size leaks a little less information overall.

We parameterize each redaction with experimental parameters of shifting scheme, font

size, font, and dictionary. We then perform the following steps:

1. Choose some word from the corpus that occurs in the given dictionary.
2. Format a PDF document containing the chosen word and surrounding text from the corpus using the parameterized font and shifting scheme.
3. Replace the chosen word at the point with each word in the parameterized dictionary (either uniformly or according to the frequency distribution).
4. Redact the dictionary word by replacing it with a glyph shift equal to its width.
5. Record the leaked information fingerprint for this dictionary word.

In summary, we simulated the redaction of all possible dictionary words at a sample of occurrences of dictionary words in the NYT corpus. We then computed the amount of mutual information leaked by the redacted document Y about the redacted word X. The amount of information leaked by the redacted document Y about the redacted word X is given by the mutual information $I(X, Y)$. Let L be a random variable representing the location of the occurrence of the dictionary word chosen in our first evaluation step, and H denote entropy. Note that $H(Y|L, X) = 0$ (there is no uncertainty about Y given L and X), since the formatting and redaction in steps 3 and 4 are deterministic, and $H(L|X, Y) = H(L|Y) = 0$ (there is no uncertainty about the location of the redaction given the document after redaction). Using these two facts,

$$
\begin{aligned}
I(X;Y) = & \\
& = H(X) - H(X|Y) \\
& = H(X) - H(X,Y) + H(Y) \\
& = H(X) - H(L,X,Y) + H(L|X,Y) + H(L,Y) - H(L|Y) \\
& = H(X) - H(Y|L,X) - H(L,X) + 0 + H(L,Y) - 0 \\
& = H(X) - 0 - H(L) - H(X) + H(Y|L) + H(L) \\
& = H(Y|L) \\
& = \sum_{\ell} Pr[L = \ell] \cdot H(Y|L = \ell).
\end{aligned}
$$

Thus, $I(X;Y)$ is the average, taken over redaction locations, of the entropy of Y, that is, the entropy of the distribution of possible documents after redaction. For a large corpus and large dictionaries, calculating this quantity exactly is expensive. Instead of calculating the exact value, we sample $H(Y|L = \ell)$ for several initial word choices.

In addition to mutual information, we calculated the probability that the leaked information can be used to correctly guess the redacted word. In the uniform distribution

Table 5.4: Number of bits leaked (left) and probability of a correct guess (right) for different shifting schemes in simulated redactions of the NYT corpus set in 10pt font.

		Leaked information (bits)									Probability correct guess								
Distr	Courier	Times			Arial			Calibri			Times			Arial			Calibri		
Dict	Mo	Un	W07	W19	Un	W07	W19	Un	W07	W19	Un	W07	W19	Un	W07	W19	Un	W07	W19
Uniformly distributed																			
Str	0.2	8.2	—	—	8.8	—	—	12.1	—	—	<1%	—	—	<1%	—	—	<1%	—	—
Acrn	0.2	6.5	11.0	9.2	6.4	10.9	9.7	11.4	13.9	13.7	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%
Word	3.3	8.7	12.6	12.3	8.7	12.5	11.8	12.8	14.3	14.3	2%	22%	19%	2%	23%	16%	16%	48%	47%
Ctry	5.0	8.6	9.0	9.0	8.6	8.9	8.9	9.1	9.1	9.1	77%	94%	93%	75%	90%	89%	97%	98%	97%
Rgn	4.1	10.7	15.0	14.6	10.2	14.3	13.6	14.1	16.8	16.7	2%	10%	8%	1%	9%	7%	3%	15%	14%
Natl	3.8	8.2	8.8	8.8	8.0	8.8	8.7	8.9	8.9	8.9	66%	90%	91%	61%	90%	88%	97%	98%	98%
FN	2.6	7.8	11.6	11.1	7.9	11.0	10.4	12.3	14.1	13.8	<1%	11%	7%	<1%	10%	6%	8%	32%	28%
LN	2.9	8.2	12.4	11.7	8.3	12.2	11.4	12.7	14.8	14.6	<1%	11%	8%	<1%	12%	7%	6%	33%	30%
FILN	2.9	8.6	13.3	12.6	8.7	13.0	12.2	13.0	15.6	15.5	<1%	4%	2%	<1%	4%	2%	2%	11%	11%
FI×LN	2.9	8.5	13.1	13.1	8.6	13.2	13.2	12.8	15.4	15.3	<1%	1%	1%	<1%	2%	2%	<1%	5%	5%
FNLN	3.2	9.4	14.5	14.1	10.0	14.9	14.0	13.5	16.7	16.5	<1%	3%	2%	<1%	4%	3%	3%	8%	7%
FN×LN	3.4	8.8	13.7	13.3	9.2	13.8	12.9	12.8	15.9	15.3	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%
Text frequency distr.																			
Str	3.0	7.6	—	—	7.5	—	—	10.5	—	—	37%	—	—	35%	—	—	74%	—	—
Acrn	2.0	6.5	8.7	7.7	6.4	8.1	7.8	9.1	9.5	9.5	44%	75%	59%	43%	67%	61%	81%	91%	90%
Word	3.2	8.1	10.9	10.6	7.9	10.6	10.2	11.2	11.8	11.7	29%	69%	63%	27%	64%	57%	74%	88%	87%
Ctry	3.0	5.6	5.8	5.7	5.6	5.7	5.7	5.8	5.8	5.8	92%	99%	96%	93%	97%	96%	97%	98%	98%
Rgn	3.1	7.4	9.3	8.9	7.3	8.9	8.6	9.6	9.9	9.9	53%	81%	75%	51%	75%	71%	88%	94%	93%
Natl	2.5	5.2	5.4	5.4	5.1	5.3	5.3	5.4	5.4	5.4	95%	99%	99%	90%	99%	98%	100%	100%	100%
FN	2.5	7.1	9.3	9.0	7.2	8.8	8.6	9.7	10.0	9.9	45%	79%	74%	46%	71%	68%	87%	93%	92%
LN	2.7	7.8	10.8	10.4	7.9	10.7	10.1	11.4	12.2	12.1	28%	59%	53%	28%	58%	51%	66%	81%	79%
FILN	2.7	8.4	12.4	11.9	8.5	12.2	11.5	12.6	14.4	14.3	8%	30%	22%	8%	25%	21%	34%	53%	52%
FNLN	3.1	9.2	13.8	13.6	9.8	14.3	13.5	13.2	15.9	15.8	4%	20%	19%	6%	24%	19%	16%	38%	37%

this is a random guess from the (typically quite small) set of matching candidate words, and in the frequency distribution we select the candidate with the highest frequency in either the NYT corpus (for *Str*, *Acrn*, *Word*, *Ctry*, *Natl*, and *Rgn*) or in the three state voter registration databases (for *FN*, *LN*, *FN×LN*, *FI×LN*, *FNLN*, and *FILN*).

Synthetic Results Table 5.4 reports on the vulnerability of excising redactions. "Probability correct guess" refers to the likelihood of randomly selecting the redacted word given the (typically small) set of matching candidate texts. The top half of the table shows the case where dictionary elements are drawn uniformly at random in step 3 above and the bottom half shows the case where dictionary elements are drawn according to their frequency of occurrence in the NYT corpus or, in the case of names, in the three states' voter registration databases. We have omitted rows for the *FN×LN* and *FI×LN* dictionaries from the bottom half because they are identical to the *FNLN* and *FILN* results,

respectively.[8]

The left side of the table shows the mutual information of the dictionary and the resulting document. This is the number of bits of information leaked by the document about the redacted word. These should be compared to the total entropy of the corresponding dictionary given in Table 5.3.

The right side of the table gives the probability correctly guessing the redacted text using only the information available after redaction. This corresponds to success probability of a game in which a player knows the dictionary and tries to guess the redacted word based information in the redacted document. In the non-uniform case, the optimal strategy is to guess the most likely (the highest occurrence frequency) word or phrase that produces the same redacted document.

The table gives statistics for four fonts: Courier, Times New Roman, Arial, and Calibri. For each font, we show the amount of information leaked by an unadjusted shifting scheme (Un), text produced using Word 2007–2016 (W07), and Word 2019–2021 (W19) dependent glyph shifting schemes. For Courier, we omit the W07 and W19 columns as monospaced fonts behave identically in the unadjusted and Word cases.

Table 5.4 does not show results for the *Str* dictionary under the uniform distribution using Word schemes because simulating redaction of all 9×10^{22} strings is prohibitively expensive. Results for the unadjusted positioning scheme were obtained without simulating redaction by exploiting the regular structure of the dictionary.

We compare redaction vulnerabilities across three categories, using the example of redacting a surname set in 10 point Times New Roman font to guide the reader:

Monospace (Mo). For monospaced font redactions, the residual information in the document after redaction reveals the number of characters in the redacted word. The Courier column in Table 5.4 thus tells us how much information is revealed by knowing the number of letters in the word. (For monospace fonts, which are always unadjusted, this is just the entropy of the distribution of word lengths.) For example, knowing only the number of characters in a surname leaks 2.9 bits of information (out of 17.2) when guessing uniformly at random from the candidate redacted texts. This has a $< 1\%$ probability of success—redactions of monospace fonts are relatively secure.

Unadjusted (Un). For independent shifting scheme redactions, e.g. a PDF produced by Google Docs, the residual width of the redaction leaks more information than the number of characters redacted. The width of a redaction of a surname (the *Un* row)

[8]We considered using the empirical distribution generated by the independent distributions of first and last names in the voter registration databases. However, the distribution of first names is *not independent* of the distribution of surnames due to cultural naming conventions.

with no glyph shifts provides 8.2 bits of information about the surname if it is chosen uniformly at random and 7.8 bits (out of 13.1) if it is chosen according to an empirical distribution. While the uniform distribution still has a $< 1\%$ probability of success, the empirical distribution has a 28% probability of success.

Dependent (W07/W19). For dependent shifting scheme redactions, the residual information after redaction leaks both width and shift equivalence classes and therefore more information. If we redact a surname from a PDF produced by Word 2007–2016, the resulting document leaks 12.4 bits of information about a name chosen uniformly at random and 10.8 bits when chosen according to the empirical distribution given by voter registration databases. With the inclusion of these information leaks, the probability of a correct guess under the uniform distribution is 11% and the probability under an empirical distribution is greater than 50%.

We found leaks of up to 15 bits of information about redacted text in dependent (Microsoft Word "Save as PDF") glyph shifting schemes. These schemes present a significant security concern for excising redactions. Even without considering the frequency of names in the population, single word redactions have a greater than 5% chance of being broken. When an adversary can use statistical likelihoods of names, two word redactions of first names and surnames face a 1 in 5 chance of being broken, and an adversary's chances are only better for other fonts.

Calibri, the default font in Microsoft Office since 2007, leaks the most information because the font has a greater variety of character widths than Times New Roman or Arial. Recall that the empirical entropy of the surname dictionary is 13.1 bits (Table 5.3), so a redacted surname set in 10-point Calibri using a Word 2007 shifting scheme leaks almost all the information available and can be correctly deredacted in roughly 81% of cases.

In summary, short (1 to 2 word) excising redactions are NOT secure. Note these synthetic attacks and our real-world attacks are performed in a vacuum, using a full dictionary of US names. In practice, an attacker can use a smaller dictionary (e.g. company employees).

Real-world Evaluation. With our synthetic evaluation done, it became immediately apparent that understanding the applicability of our simulations to real documents was critcial. The generality of the results not immediately clear because glyph shifts may be modified by a variety of software workflows.[9] It is also unclear whether there exist a

[9]For example, by opening the PDF produced by word *and then* modifying it using Adobe Acrobat.

significant number of vulnerable redactions in real documents.

We consider both *nonexcising* and *excising* redactions in this section. We chose to study redactions of names (e.g. first names, surnames, and country names) in this section because they are the most common, discussed further below, and because their release presents a privacy concern. We do not release any of these names and have taken steps to notify affected parties in order to ensure no individual will be adversely affected by the present analysis (Section 5.5).

We evaluated the following document corpora:

1. *FOIA.* Documents obtained via the US Freedom of Information Act (FOIA) on governmentattic.org [40]. This corpus provides us with independently selected documents with some public interest.

2. *OIG.* Office of the Inspector General (OIG) reports hosted by oversight.gov [39]. The OIG is a US Government oversight branch tasked with preventing unlawful operation of other government branches. This corpus allowed us to measure the impact deredaction may have on documents from a high-profile and large organization.

3. *DNSA.* Digital National Security Archive (DNSA) documents produced after 2010 [37]. The DNSA is a set of historical US government documents curated by scholars. That is, we found redaction information leaks affect significant historical documents.

4. *RECAP.* CourtListener's RECAP court document archive. RECAP mirrors PACER, the US Federal Courts' docketing system [38], and contains over 10 million documents. We use RECAP to measure the impact of nonexcising redactions (discussed below).

5. *rRECAP* the subset of RECAP documents returned for the search string "redacted". We chose to include rRECAP because running the excising redaction location algorithm mentioned on the entire RECAP corpus was both computationally and financially prohibitive.

Only the RECAP corpus contained nonexcising redactions and our results for this corpus are reported with respect to nonexcising redactions. Our results for all other corpora are reported with respect to excising redactions.

These corpora provide a sample of vulnerable documents and are not intended to be comprehensive analyses of any single affected party, e.g. government agency.

We also chose to restrict our evaluation to first name and last name redactions. In a sample of 100 redactions from our corpora, 52 were personal names (determined manually based on the surrounding text), 26 were multi-word phrases too long to attack, 17 were numbers (not vulnerable to attack), 3 were pronouns (trivial to attack), and 2 did not have identifiable semantics. We included titles, (Mr., Mrs., Ms., and Dr.), initials, such as "J." and "S.", and possessive forms, such as "'s", in our dictionary. However, our results for matches *remove* these modifications, so "Ms. Doe", "J. Doe", and "Doe" count as a single match.

Our final dictionary contained 7,066,800 entries. We do not include $FN \times LN$ as testing this dictionary takes 6 hours on an Intel Xeon Silver 4208, 2.10 GHz, 32-core server and the result sets are typically large.

The only exclusion we made for this dictionary restriction was in our evaluation of the DNSA. The redactions matching the Microsoft Word dependent glyph shifting scheme in the DNSA were of an acronym, demonyms, and countries.

We evaluate a redaction if:

1. The redacted text is present in the PDF (vulnerable to copy-paste attack); or
2. The redacted text is not present, but the document retains glyph shifting scheme information where:
    - The scheme matches a Word "Save as PDF" shifting scheme,
    - The redaction appears to be a name, e.g. "Jane", and
    - The redaction is the first from left to right on the line of text.

These criteria provide a uniform and accurate proof-of-concept evaluation of the impact of the discovered redaction vulnerabilities.

Real-World Evaluation Limitations. We chose to restrict our attacks to the Microsoft Word schemes because these leak the most information, and are thus of the greatest concern. Other shifting schemes exist and evaluating these schemes' security will require further reverse engineering. Cases matching the Microsoft Word "Save as PDF" workflow represented 8.8% of redaction instances across all four corpora. We also considered only the first redaction on a line to remove any dependence on prior correct guesses, which would be necessary to attack the second redaction on a line.

Deredaction also requires modeling the glyph positioning scheme of the PDF document and identifying a dictionary of possible redacted texts. Despite some scripting possible using time-travel debugging, at the time of writing, this was a manual process. Therefore, we report the conservative *Evaluated* row, which are redactions for which we were able to perform an automated attack given our modeled Microsoft Word positioning scheme.

We also took three steps to ensure the results of our experiments were accurate:

1. We required all potential deredactions to exactly match all present PDF glyph positioning information.

2. For excising redactions, it is necessary to identify the shifting scheme used by the PDF document. This process avoids incorrect matching and filters out regions of documents with idiosyncratic formatting. For example, Microsoft Word documents may have alternative layouts for text, e.g. text boxes, tables, and line numbers.

   In the present analysis, we considered a PDF page to *match* a scheme if we identify greater than 100 glyphs with shift values matching the scheme on the page. We only count entire lines: all of a line's glyph positioning information must match the scheme exactly to count toward the 100 glyph threshold.

3. We manually classified each excising redaction as being a redacted name based upon the content of the surrounding text. We were conservative with our classification and only attack redactions we are *certain* are of names.

4. Where possible, we validated our findings using public information.

We also identified redactions in documents using unadjusted and Adobe OCR glyph shifting schemes, though did not attempt to deredact these cases. These schemes leak 2–3 bits less information than documents produced by Microsoft Word.

Finally, To make sure we were not missing any PDF documents originating from Microsoft Word, we also counted redactions within a 10 text space unit L_1 distance from our Microsoft Word model. These PDFs were the result of a non-standard workflow, e.g. creation using Word 365, the web interface for Microsoft Word. In our results, we label these redactions as *Near Word*.

Real-world Results The evaluation was able to find redacted information of significant public and historical relevance (not included). We begin by briefly discussing the non-excising redactions we located. We ran Edact-Ray's location algorithm for nonexcising redactions on all of RECAP ($\approx 10^7$ documents) and found 6,541 nonexcised redacted names in 710 US court documents. The number of total nonexcised redacted words was larger ($\approx 1.4 \times 10^5$). Of the redacted names, 327 had a independent glyph positioning scheme, 445 used Adobe OCR, 20 were close to Word in L_1 distance, 5,691 were not positioned using a scheme we modeled, and 58 exactly matched the Word positioning scheme.

Table 5.5: Potentially vulnerable redactions and glyph shifting schemes identified in redacted corpora pages. The "Evaluated" row is filtered for the length of the redaction (approx. 1 to 2 words), semantic context (being a name), and scheme (Word).

Metric	FOIA	OIG	DNSA	rRECAP	RECAP
Documents	3,145	1.9×10^4	678	2.5×10^4	$\approx 10^7$
Pages	4.9×10^5	5.2×10^5	1.2×10^4	3.4×10^5	$\approx 10^8$
Redacted PDFs	236	1255	7	67	710
Redactions	4.5×10^4	1.3×10^4	235	1,221	6,541
Unadjusted	2,844	314	7	7	327
Adobe OCR	3,406	1,814	0	224	445
Near Word	175	114	3	33	20
Unrec.	1.3×10^4	1×10^4	214	838	5,691
Exact Word	4,694	455	14	119	58
Vulnerable	711	58	9	0	58
No Matches	382	39	1	–	N/A
Uniq. Matches	3	0	5	–	N/A
Avg. Matches	2,435	4,260	393	–	N/A
Med. Matches	494	1,081	1	–	N/A

Due to the computational cost of locating excising redactions in the *whole* of recap, we did not attempt to evaluate excising redactions for all US Court documents, though we confirmed some historically important court documents are affected. We also found *no* nonexcising redactions in our other corpora (expected, as we were informed they have significantly more robust redaction workflows).

Table 5.5 also reports our findings on the security of excising redactions. The *Evaluated* row reports the result of our redaction classification methodology. We identified 711 immediately vulnerable excising redactions in the FOIA corpus, 58 in the OIG corpus, 9 in the DNSA corpus, and none in rRECAP (mentioned above).

We deredacted the FOIA and OIG cases using the full cross product dictionary $FN \times LN$, 1.5×10^{10} entries. For FOIA, we had three cases of a single, unique name being returned as the redacted text, 2,435 possibilities returned on average (a 3,000-fold reduction), and 494 on the median. For OIG, we had no cases of a single, unique name being returned as the redacted text, but 4,260 possibilities returned on average, and 1,081 on the median. After, we also validated many of these cases by looking up document details using google, and found an empirical distribution of name probabilities returned the correct deredacted name in several cases. The DNSA redactions were all of demonyms, and used the 509 entry dictionary. This resulted in 5 cases of a single, unique name being returned as the

redacted text, with 393 possibilities returned on average, and 1 match on the median.

We find unmatched cases are common (382 cases in FOIA, 39 in OIG, and 1 in the DNSA). Manual analysis of nonexcising RECAP redactions found 28.2% of names were of a form occurring in our dictionaries, leaving 71.7% in other forms (e.g. first name, last name).[10] Our own unmatched case rate (54.7%) was lower than expected, and indicating matches were reported but the name was not in the dictionary in approximately 17% of cases.

Real-world Validation As was noted in Chapter 3, any guess and check performed for an unknown peice of information should be understood as *ruling out* possible candidate texts rather than determining exact content. To serve as an oracle and ensure our methods were correct, we therefore also manually validated our results for excising redactions (where possible on 8 PDF documents) by performing web searches for further information on the redacted document. For example, in the case of an OIG investigation, we would perform a search related to the offense committed and organizations affiliated. In this process, we found several cases where Edact-Ray's returned matching name set included the ground-truth name. We did not find any cases where a matching set was reported and Edact-Ray did not return the correct name as one of the results, though these cases likely exist.

Nonexcising redactions provide ground truth, and we also used these redactions to validate our techniques. For every nonexcised redacted name in RECAP present in our dictionary, we excised the name, i.e. we removed the redacted text but preserved non-redacted glyph shift information. In all cases, the set of candidate texts returned by Edact-Ray included the ground truth redacted word.

5.5 BROADER IMPLICATIONS

With the conclusion of this discussion on simulation for the purposes of analyzing redactions, we may look towards the broader implications of this study on the topics of this dissertation. Although the subject of this dissertation is on the construction of effective emulations for use in security measurement, it also is pertinent to discuss, briefly, the ethical implications of this work.

[10] We chose to evaluate the simplest formatting ruleset possible for simplicity.

Ethical Implications The application of effortful analysis to this subject demonstrated sub-pixel-sized glyph position shifts, imperceptible to the human eye, can break text redactions. There are at least 778 vulnerable excising redactions in FOIA, OIG, and DNSA PDF corpora and more than 700 publicly accessible court documents with nonexcising redactions. As a result, it is necessary to discuss the ethical implications of this work and potential defenses.

The official NSA guidelines for redaction of Microsoft Word generated PDFs are to change the content of the original Word document so that information regarding the sensitive name is destroyed (i.e. by changing the name to the letter "x") [101]. This is likely the only *perfectly secure* manner by which to protect a redaction from information leaks. However, this can be an invasive process and is not always applicable to old documents for which the original files have been lost. Below, we describe five alternative defenses against excising redaction vulnerabilities, which help to make redaction leaks more difficult to exploit:

1. **Glyph Shift Discretization** Shifting scheme noise may be added to a line without affecting visual fidelity. Alternatively, each shift could be rounded to a discrete interval, e.g. 0.1 mm. If this noise is indistinguishable from legitimate glyph shift information, accounting for it would require an adversary to increase the set of accepted redacted text guesses. Removing shifting scheme information altogether can also lower information leaks, though this is less visually appealing.

2. **Document Layout and Redaction Obfuscation** Modifying the PDF commands used to render the box of the redaction complicates the process of automated redaction location. For example, our excising redaction location algorithm relies on identifying a black box between two US English words in order to avoid large numbers of false positives.

3. **Increasing Redaction Width** Redacting additional adjacent words can make deredaction more difficult, although this practice may not always compatible with legal mandates. This defense increases the number of words the attacker must guess. If the adjacent words are easy to infer, e.g., by a machine learning model, this is not a sufficient defense.

4. **Adversarial Redactions** One defense against deredaction is to change the document's text before it is redacted, for example, by replacing a sensitive name with the letter "X". It is also possible to *lie* to deredaction by changing the redacted content to something seemingly valid, potentially misinforming an adversary.

5. **Rasterization** Rasterization appears to be an effective defense against deredaction. In many cases this defense is infeasible because it removes searchable text data from the document, however, performing OCR on the document post-redaction can act as a stopgap for this issue. Rasterization algorithms may also modify or ignore certain glyph shifts,[11] requiring the analyst to perform more reverse engineering to identify the specific rasterization tool used.

As it is common practice to rasterize documents, we performed an experiment to estimate the effects of rasterization on redaction security. We chose ten name occurrences from court documents using the Word shifting scheme in 12 point Times New Roman. For each selected text line, we substituted the name with each entry in our 235,560 name dictionary from our real-world evaluation, redacted the entry, and calculated the redaction's width. This resulted in 2,017 unique redaction widths, ≈11 bits of information leaked, on average. Quantization to 300 DPI (20 text space units) resulted in 252 widths on average (≈8 bits) and quantization to 600 DPI performed slightly worse with a result of 377 widths (≈8.6 bits).

Unfortunately, this estimation is a lower bound. In general, when a character is rasterized, the vector representation is converted to a "mosaic" of pixel values. Whether a given pixel value is black, white, or in the case of anti-aliasing [102], some shade of gray, is dependent on how the graphical rendering specification is converted to a matrix of pixel values. An analysis of this possibility is beyond the scope of this dissertation.

In addition to providing access to some of these defenses publicly, we carefully considered the balance between the benefits of public awareness and potential risks of misuse present in this research. To the best of our abilities we are attempting to maximize the benefits to society and minimize the harm to individuals in the publication of this work. All the documents studied are in the public domain. So long as copies of these PDFs exist, they pose a risk to individuals' privacy. During our evaluation, we ensured all deredaction was performed on an isolated and hardened server and that no identifying information exited this server. We deleted the documents from our server and associated data after evaluation and notifying affected parties.

We have notified Microsoft and Adobe of our discoveries: they have acknowledged the attacks are possible, but to our knowledge have not yet made changes to their tools that could significantly help protect future PDF redactions. Our discussions with government officials indicate remediations will require both redaction *process* and *software* changes. We have also reached out to the PDF Association regarding the provision of guidance for

[11] We analyzed several rasterization tools, finding several had imperfect precision.

redaction application implementers. We have also performed an extensive notification of the US Courts, the Free Law Project, the US Department of Justice, and 22 agencies affected by our study of documents from the Office of Inspector General. The US Courts, US Census, and CIGIE have been active in both understanding and remediating the problem. Notification has been given to sub-courts affected by nonexcising redaction vulnerabilities. Our discussions with these groups are ongoing and include technical support, code, and free consulting regarding remedation and prevention efforts.

The discovered redaction vulnerabilities are not limited to US documents and affect any PDF document positioned using non-uniform font metrics (whether these are Word-internal metrics, TTF metrics, or otherwise). Due to the limitations on the author's knowledge of different languages, we were not able to effectively evaluate the security of documents internationally. We have indicated to US government officials that notification of the international community is advisable.

Implications for Security Measurement Our study of the simulation of glyph shifting schemes in this section point towards several critical limitations in the study of systems generally. First, the importance of limiting the existence of *cartesian doubt* in the determination of information by ensuring that the dictionary selected as a basis for further deduction is sound. Second, this section clarified the possibility, in the case of some *oracle* to determine whether a guess is correct and a reasonable search space, it is possible to attempt a universal quantification over the possible inputs of a simulation in order to find inputs that "work". In cases where the amount of information leaked by a system is large enough, this is sufficient to determine the missing component. Finally, the danger of ignoring hidden dependencies in information about a system: this is captured by the formal study of *side-channels* [103] in hardware and cryptography, but is not always applied evenly as a concept across different security-sensitive systems. To provide a concrete example, we will note prior work on redaction:

The primary predecessor to our redacton work is Lopresti and Spitz [104], which presents a manual technique for matching glyphs to a redaction's width in a raster image of text. This attack is prone to human error, especially in light of the fact some glyph shifts are sub-pixel sized. The authors attempted to use natural language processing to predict redacted words, something our work found imprecise given current models. The Lopresti and Spitz work also conflates document glyph position specifications with TTF glyph widths and assumes both are equivalent to a raster document's character widths. This presents two problems:

First, a rasterization workflow may change a document's glyph positioning and physi-

cal printing may not be a pixel-perfect reproduction of the digital document. While the accurate representation of documents is the whole *point* of PDF, a few of our tests found glyph positioning operations were not always honored. Behavior and information from any *singular* tool may or may or may not comply with the PDF standard, either because of software bugs or due to a lack of support for a specific glyph positioning operation.

Second, TTF glyph widths do not necessarily equate with PDF document or raster glyph widths. TTF is only one of five types of fonts supported by PDF. The Lopresti and Spitz techniques also rely on (potentially inaccurate) human determination of glyph positions. Recall the present work provides a fully automatic deredaction method, a precise analysis of leaked information, and a clear measurement of this problem's prevalence in real documents.

Whelan and Naccache [105] also performed the first case study of width-based techniques for deredacting documents. They placed images of unredacted glyphs from a printed document into the space of a redacted country name in an Iraq War memo. While of historical importance, Whelan and Naccache's technique is manual and error-prone for large dictionaries. As shown by our evaluation, the width of a glyph is most often *not* independent of its location on the page.

While the above had some idea of how attacks on redactions may work, their analysis did not present a principled study of the encoding of redactions in documents or simulation of PDF document production. As a result, this prior work did not accurately measure the security of redactions, demonstrating how a lack of proper measurement and verification of common systems can lead to disaster.

5.6 SUMMARY

In this chapter, we have presented a principled study of the security of redactions in PDF documents. We presented techniques and methods for creating simulations of complex software functions using time-travel debugging and manual analysis. In doing so, we also demonstrated a unique and impactful application of dictionary attacks in the domain of digital forensics. Insights from this work can be generalized to many scenarios where some information about the system under study is missing. Moreover, this was the first work to demonstrate effective attacks on a large number of excising redactions—redactions where character information is removed. Our empirical results show that the attacks are effective on a large number of documents, including those from the US government. This led us to notify more than 22 affected parties, including large

government organizations, and release effective tools for the protection of redactions. Finally, we related this work to the primary subject of this thesis: redaction vulnerabilities present a case study demonstrating the importance of accurate simulation when evaluating a system's security.

Chapter 6: Logic-based Function Extraction

In the approaches to emulation-based measurements of system security we have considered thus far, the primary limitation has been the representation of missing information. In real-world systems, components are often only partially accessible due to a mixture of closed-source code, limited physical accessibility, or intractable complexity. Traditionally, the extraction of clean mathematical models from systems, such as ordinary differential equations, is considered infeasible. Simultaneously, a significant amount of work has now been performed on the verification of models with black-box components, such as DryVR [106]. In this chapter, we will discuss an alternative method of verifying complex, real-world systems which works by extracting key information from a given complex system's binary representation. We will introduce the InteGreat system [11], which uses a combination of a logical rule specification framework, program slicing, and abstract interpretation to lift high-level emulations (continuous equation models which can be dropped into Matlab Simulink). In the latter sections of this chaper, we will demonstrate how this approach can be used to identify differences between implemented machine code and published mathematical specifications.

6.1 INTRODUCTION

Verification of cyber-physical systems (CPS) is a challenging problem, often simplified by the treatment of complex implemented components as black-boxes. When considering the system itself, mathematical expressions for these systems must often be translated into low-level languages (C, Ladder Logic, etc.) and compiled for use in a physical system. The resulting binary code may also be symbol-stripped and lose information, e.g. the names of variables, present in the original control algorithm. In many cases, however, because the system retains something of the high-level semantics, it should still be possible to recover the original high-level model intended for a verification system. As an added benefit, by working directly with firmware, we could verify and analyze programs where no high-level specification has been written.

While tools such as Ghidra [45] are capable of lifting a binary to pseudo-C representations, these outputs cannot be translated into a more abstract representation of the program. Adapting these existing tools to lift to a given higher-level representation also requires significant effort, as their internal representations are static and aimed primarily at capturing microarchitectural semantics. However, if natural deduction could be ap-

plied to express the nature of the semantic patterns as they appear in the binary, then lifting to a particular abstract representation could be made modular and portable. This presents two particular problems of extraction and incorporation: in the former case, the ability to derive an abstract semantic from machine code operations, and the ability to incorporate outside information into the lifting process.

The current state of the art in function extraction involves the use of binary symbolic execution to derive a model of the binary's code over the theory of bitvectors, as was performed in part in Section 4.2. This approach deduces a mathematical model of a computation's behavior, but does not provide any framework for encoding common rules of abstraction. Moreover, no current binary symbolic execution systems do not provide any means of precisely defining slices of the program to serve as boundaries for the lifting process, and rely on the user to isolate and substitute any subtrees of the AST for simplified expressions. The result is a set of systems that could theoretically serve to automatically lift a binary to a higher-level representation based upon logical rules, but which in practice lack much of the necessary automation.

We therefore introduce InteGreat, a system which takes in a set of logical lifting rules (provided via Z3 [76]) and outputs of symbolic execution to translate closed-source, symbol stripped binary firmware into verifiable mathematical models. We achieve this by allowing users to nest and chain rules for lifting and decompilation in a natural, declarative syntax, allowing for the modular capture of complex semantic operations. For example, we may associate a sequences of calls to multiple different child functions as a single operation and we lift a sequence of non-adjacent low level instructions into a single function call using the same effective syntax. To our knowledge, this is the first lifter to use explicit logical propositions in targeting the correct abstract domain representation of a binary program.

InteGreat significantly extends the angr [107] symbolic executor to bind the semantics of analyzed program slices into meaningful abstract symbols, replacing complex inner computations with a uniform representations, while maintaining the integrity of the symbolic state's semantics. In particular, we focus on lifting to continuous equations that may be "dropped in" as Matlab Simulink function blocks [42] for evaluation in the context of an environmental model. Under the system's default operation, a user only needs to specify entry and exit points in the firmware binary, and the system will replace function calls with uninterpreted functions, lifting sequences of otherwise complex operations into a single symbolic value.

To demonstrate the capabilities of such a system, we use InteGreat to show how logical rules can be used to lift continuous equations from a quad-copter firmware implement-

ing Sebastian Madgwick's Orientation Filter [108, 109], and demonstrate these lifted equations uncover an implementation error in the quad-copter firmware, finding a three-way difference between the stabilization algorithm's mathematical representation, C, and third-party C implementations. We also analyze a PLC firmware regulating a chemical plant's reactor pressure [110] and use the equations recovered by InteGreat to determine the sensor inputs necessary to precisely destabilize the reactor pressure of the chemical plant and exactly reproduce a code upload attack performed by prior literature. The lifted representation of the chemical plant's firmware also communicates a subtle discovery: it is not possible to automatically infer representations of analog-to-digital unit conversions at I/O boundaries without additional information from a model of the physical environment.

6.2 RELATED WORK ON LIFTING

This work brings together multiple methods from a rich history of verification of software and firmware systems. While our discovery, the application of logics to the domain of decompilation and lifting, is novel, we are indebted to prior work for the introduction of nested abstractions, abstract interpretation, program slicing, function summarization, and many other techniques:

- Currie et al. [111] use the substitution of program slices with uninterpreted functions to determine program equivalence across compiler optimizations. However, they do not address the idea of using nested logical rules to perform lift binary programs or the complexities involved in stitching together abstracted program slices. We provide a total framework for retaining viable semantics before and after skipping a program slice, even at the bitvector level. In fact, Currie et al. note that there may be ways to improve the accuracy of their matching by "combining uninterpreted functions with a bit of bit-level analysis."

- Sery et. al [63] leverage the FunFrog system [112] to extract function summaries after an initial verification run using source-code level symbolic execution to recover a Bounded Model Checker (BMC) logical formula representing the original program. The work performs AST slicing and variable binding methods similar to our own and uses the Craig interpolation (the identification of a sub-proposition that implies two other statements) to refine their function summaries. In contrast, InteGreat targets firmware and thus supports the abstraction of micro-architectural semantics.

We also introduce the idea of using logical formulae to give rules for the translation rather than strict substitution of abstracted program slices based upon the semantics of the slices themselves (interpolation).

- A line of work has been done using symbolic execution to perform model extraction and subsequently verification on the extracted models. SPIN [113], defined the term "model extraction" and applied model-checking on aero-space flight software. Babić and Hu [114] used natural abstraction boundary identification and symbolic execution to optimize the performance of verification. Hernandez et al. [83] and [115] used symbolic execution to extract and verify protocol models. The same authors also noted the importance of rounding the floating-point precision error on verifying their extracted models in [116]. Jackson and Woodward [117] extracted object-oriented (OO) models from Java byte-code, [118] extracted OO data models from weakly-typed source code. Bandera [119] is a tool for user-guided extraction of finite-state automata from Java programs, however, the tool requires access to source code, and focuses on abstracting single variables rather than program slices. While all of these techniques could improve InteGreat, prior work does not address the possibility of a generic framework for the specification of lifting operations, and does not solve the specific problems involved in stitching together uninterpreted functions as abstractions. Our work also does not rely on static matching of semantic patterns and supports nested and chained natural deduction rules for lifting.

- Ji et al. [120] perform backward application of extended sequent calculus rules on symbolic expression trees. Our use of nested logical statements is similar to sequent calculus, however, the goal of Ji et al. was the bisimulation and optimization of the analyzed algorithms using sequent calculus, not to perform lifting.

Ultimately, we were somewhat suprised by the lack of a work approaching the subject of extracting functions from a system using logical operations. However, this use case was also motivated by the specific insights introduced in the emulation and modeling of highly complex systems (Chapters 4 and 5). The InteGreat system draws on the specific challenges of the Jetset and Edact-Ray systems to inform its unique approach to the modeling of systems.

6.3 DESIGN OF A LOGIC-BASED LIFTING SPECIFICATION

We begin our design of InteGreat with program slices, introduced in Section 2.1.2, which we denote with γ, and the location of which we denote f. By default we resolve each γ to a function call boundary, but the determination of each γ's location is also user-scriptable via a provided wrapper around Ghidra's static analysis APIs. We enforce that the control flow of each slice must have either a total or no overlap with any other slice (an independence system).

As InteGreat runs, it begins analysis at the innermost γ and works outward, identifying all the input and output locations (side effects) of each γ via symbolic execution. By default, these side effects are used to abstract the semantics of each γ into a single uninterpreted function mapped to f (the location of the call). Specifically, we perform three actions upon reaching a location f for a given γ:

1. We associate f to a set of logical formulae for lifting rules ϕ, potentially input by a user, providing a deductive sepecification for *how* to lift f. For example, encoding the logic of splitting a floating point value across r0 and r1.

2. We associate f's symbolic state with the set of input locations of γ: every potential register and memory location used by the output constraints provided by a full symbolic execution of γ. These are then available as symbols in ϕ to determine the appropriate parameterization of f.

3. We associate encountering f with a specific timestamp to ensure the lifted representation respects the program's order of operations.

Our framework provides prospective solutions to the problems of unconstrained pointer values and loop invariant inference during symbolic execution. To cope with unconstrained pointers we represent each read or write to memory with both the value and the symbolic expression used to determine the address, and enforce a strict equivalence between the two (discussed next). We expect a zero-length program slice and associated ϕ to be defined which can be used to resolve any runtime-ambiguous values of pointers. For many problematic loops, we can define two program slices: an outer which captures the formulae for handling the loop guard condition and an inner which captures the loop's body expression. If suitable, we may then replace the guard condition with a concise symbolic represenation, e.g. Σ, and the inner with an inferred representation of the effects of the loop body as it relates to the outer operation.

As a fallback for fully automatic use of InteGreat, we accept the traditional symbolic execution search strategy of unrolling loops and provide a few sensible defaults for the

search strategy, e.g. taking the state that generates the most complex set of constraints or the largest number of writes to memory.

Once the specific challenges of potential state explosion are dealt with, each γ is associated to a new symbol $s_0, s_1, \ldots$ and $\phi_0, \phi_1, \ldots$ to perform a translation into a new, higher-level language. These formulas are also *nestable*, and rules may be written which take some s and ϕ as input to produce a second s' and ϕ'. For InteGreat's evaluation, we introduce ϕ and γ such that unmanagable machine code is be abstracted into a Matlab-identical representation inside Z3. Thus, theorem prover expressions serve as a flexible intermediate representation throughout the lifting process.

Symbolic Memory In most firmware programs, inputs and outputs are context-sensitive to its program state, for example, the dynamic execution stack, or a dereferenced pointer to a memory location. It is thus necessary for InteGreat to allow symbolic expressions as memory addresses via a *LOC* function.

InteGreat accomplishes this by implementing a symbolic memory model, extending Trtík et al.'s model [121], under-constrained memory [122], and incorporating the core ideas of Coppa et al. [123]. Leveraging the angr event hook infrastructure, InteGreat intercepts all program state modifications, and integrates them with effective address use tracing in order to identify *aliasing*, which occurs when two symbolic pointers are used to reference the same location but their specific value is runtime dependent. With each pointer dereference, the symbolic memory model is queried with the intercepted dynamic symbolic state of the symbolic executor, which handles returning the correctly intended high-level variable value.

Pointer reasoning is provided by the equivalence of symbolic expressions for pointer values. When resolving memory, we use timestamps associated with each memory operation and a strict equivalence check on the pointer expression to determine when a memory location is modified. For example, given two pointers p_1 and p_2, if $p_1 = 50$ and $p_2 = 50$ at runtime and we are checking $read(p_1) = write(p_2)$, we will return that they are NOT the same memory location. That is, we do not assume runtime knowledge. We assume that if such knowledge is needed, a zero-size program slice (an entry point address equivalent to the exit address) will be defined that will identify this runtime equivalence, and other tools exist for this [124].

Correctness Traditional views of compiler (and by extension, decompiler) correctness involve ensuring uncompiled programs, interpreted over an input, result in the same output as compiled programs interpreted over the same input. In the bottom-down

view of compilation from a language operating over $\mathbb{R}$, we already implicitly accept the imprecision introduced by compiler/ISA when equations were compiled are executed. One argument for the correctness of InteGreat's approach is that we may implicitly accept the imprecision introduced by the inverse operation: lifting from the ISA to the real domain semantics.

However, this does not deal explicitly with the information lost in InteGreat's approach to lifting, as features from the initial domain $\mathbb{D}$ may not be present in $\mathbb{D}'$. In our earlier example, we mapped from an operation over a Galois field of 64-bit floating point values to an uninterpreted function over $\mathbb{R}$, and the semantics of the Galois field are not preserved in the uninterpreted function. This trust in the decompiler is further strained when we begin to talk about abstracting arbitrary, complex program slices. For example, the compiled version of a derivative function may be entirely different in implementation than a Matlab continuous domain equivalent.

We therefore rediscover a common solution to the problem of determining equivalence between complex computations: correctness is ensured by a reduction proof from the original function to the abstract function given a configurable error bound on the difference between outputs in $\mathbb{D}$ and $\mathbb{D}'$, ε. A similar approach was outlined in Section 3.4. The structure and representation of this proof is flexible and domain-specific, so long as it incorporates Φ and guarantees the limit ε holds. Proofs must be performed on a case-by-case basis as the non-abstracted function and targeted domains can take any of infinite forms. For example, a proof for a straight-line program performing floating point operations could consider the error introduced by each operational step in the concrete case, and then chain these values together in order to derive a total acceptable bound for ε. In the present chapter, we forgo a general proof and demonstrate equivalence empirically.

InteGreat's Interface *Sufficient input to InteGreat is a program binary, an entry point at which to begin symbolic execution and a list of start and end addresses of program slices.* Slices γ take the form of a range of addresses (potentially of zero length), a type of instruction, or an expression over the names of other slices. To avoid repeating the same ϕ for multiple γ referring to the same operations, we allow multiple range specifications for a given slice. This input is provided by a series of files which name each γ and provide Φ via a *binds* array of assignments where the left hand side is a symbol name and the right hand side is a Z3 expression.

Binds is initialized with the following two symbolic name sets:

1. The inferred inputs of γ, e.g. i0, i1, . . . ,

Algorithm 2 Abstraction Lifting

Input: γ, *binds*
Output: $A^f_{\Phi(P)}$

$A^f_P \leftarrow SE_{\mathbb{D}}(\gamma)$
for each i **in** inputs(A^f_P) **do** // Bind input to expressions
    $binds_i \leftarrow READ(LOC(i))$
end for
for each o **in** outputs(A^f_P) **do** // Bind abstract expressions to output
    $E[outputs_o] \leftarrow \Phi_\gamma(o)$ // Apply deduction rules
    $v \leftarrow new\_sym\_name()$
    $binds_v \leftarrow E[outputs_o]$ // Associate name to expression
    $WRITE(LOC(o), v)$ // Write abstraction to symbolic memory
end for
for each b **in** *binds* **do**
    $binds_b \leftarrow \Phi(b)$// Common axiom support
end for

2. Any prior expressions in the symbolic executor's state prior to reaching γ, and (optionally)

3. The full *inner* expression sets for γ' inside of γ before and after the application of the ϕ' associated to the specification of γ'.

(3) is equivalent to preserving in memory the outputs of prior InteGreat lifting steps, allowing us to nest lifting rules in a modular fashion, and (2) allows us to "chain" lifting rules such that, given γ' following γ, symbols returned by γ can be used to reason about the lifting of returned by γ'.

The output of InteGreat is a simplified Z3 AST corresponding to the application of the lifting rules. In the evaluation of InteGreat, we structured program slices and inference rules such that translating the Z3 AST to Matlab was trivial. However, the generic structure of Z3 as an intermediate representation makes it suitable for other tasks, e.g. inferring fixed points and loop invariants for lifted code [125].

InteGreat's Algorithm for Lifting Before describing the InteGreat's lifting algorithm, we introduce some notation. Let A^f_P to refer to the AST returned by angr prior to the application of Φ, similar to Def. 3.3.2, and E to refer to an intermediate storage location for Z3 AST expressions. Φ_γ refers to the set of logical statements registered for γ, $\Phi_\gamma(o)$ refers to the full set of statements used to derive o in Φ from the *binds* array. Recall that we also implement a symbolic memory addressing system for unconstrained pointer

values; we represent locations (either in registers or memory) through the *LOC* function. Note that *READ* and *WRITE* operate on A_P^f to produce $A_{\Phi(P)}^f$. $SE_{\mathbb{D}}(\gamma)$ refers to an augmented form of symbolic execution of γ using angr in the domain $\mathbb{D}$.

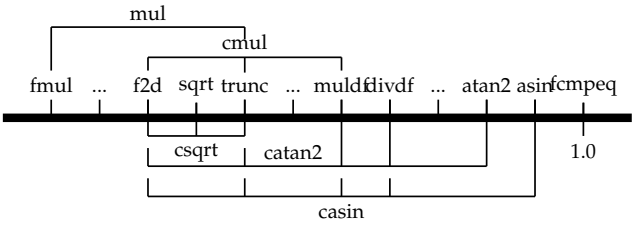
Algorithm 2 presents InteGreat's core lifting mechanics. As noted, the input to the algorithm is a program slice γ and an existing set of symbolic bindings *binds*, and the output is a new AST $A_{\Phi(P)}^f \subseteq \mathbb{D}'$, representing the lifted program. When SE encounters an inner γ', we look up the associated specification for γ' and apply Alg. 2 recursively to γ' and propagate our results to the current symbolic state appropriately.

Line 1 retrieves a set of registers and memory locations $inputs, outputs$ describing the micro-architectural input and output locations of a program slice. These are provided via a symbolic executor with modifications to support the use of symbolic values as memory addresses. For locations with existing symbolic expressions, InteGreat's symbolic memory dynamically fetches the most recent written expression, or else InteGreat writes and returns a fresh symbolic variable in that location (the *READ* function).

For values stored in memory, InteGreat dereferences the input locations to the correct symbolic expressions representing their values in InteGreat's symbolic memory model (discussed next). These symbolic values are then assigned into the inputs symbols i0, i1, ... of the *binds* array (lines 2–4). Note that the inputs may be sets of previously lifted abstractions.

InteGreat then replaces the existing lower-level, concrete expressions returned by symbolic execution for the slice's outputs with the higher-level abstraction (lines 5–10). For example, consider a floating point multiply over registers r_0 and r_1, with output in r_0, with a simple ϕ that returns an uninterpreted function f over the inputs. A fresh symbolic variable v would be created and written to the symbolic memory model, with the concrete register location r_0 storing $r_0 = v \mapsto f(i_0, i_1)$ rather than $r_0 = i_0 * i_1$. This step also introduces fresh symbolic mappings to expressiosn in the *binds* state. All future symbolic execution and calls to Alg. 2 will now operate on the abstractions provided by *binds* rather than the original concrete symbolic expression for γ.

Lines 11 through 13 of Alg. 2 provide support for common axioms. The full set of Φ for γ are reapplied at the end of analysis to resolve existing AST features into higher-level abstractions. We found this useful for lifting certain common operations across different slices. For example, we created rules for treating certain type-casts as equivalent, e.g. when a double is rounded to a float or a float is extended to a double, the Galois field variable should still refer to the same lifted variable over $\mathbb{R}$.



(a)

Version	#Lifts	#Symbols	#LoA
Firmware	335	151	728
Corrected	343	149	730
Published	283	159	637

(b)

Figure 6.1: **(a)**: Depiction of hierarchy of lifting rules for a quad-copter stabilization algorithm. **(b)**: Quad-copter firmware analytics, broken down into the number of program slice locations lifted, unique symbolic variables used, and lines of assembly code (LoA).

6.4 EXPERIMENTS ON A QUAD-COPTER AND PROGRAMMABLE LOGIC CONTROLLER

This section presents the evaluation of InteGreat on a quad-copter firmware and a PLC firmware stabilizing the reactor pressure of a chemical plant. In particular, we investigated a few concrete questions regarding the possiblities of InteGreat's design. We first consider whether, given the correct Φ, InteGreat can correctly lift machine code programs to the continuous domain. Next, we determine whether InteGreat's approach can help identify inconsistencies between published results and different implementations of the same algorithm. Then, InteGreat's lifted representations are testing in the analysis and discovery of novel features in a Progammable Logic Controller. Finally, we consider whether it is possible to correctly reproduce idealized models and attacks on cyberphysical systems using InteGreat's lifted representations.

To answer these questions, we evaluate InteGreat on two real-world embedded systems: a quad-copter firmware and a PLC firmware. We considered the firmware for both of these systems to be closed-source and symbol-stripped: the inputs to InteGreat are not dependent on any meaningful variable names or function names. We also not that the firmware was not obfuscated, and that the firmware is not using any anti-analysis techniques.

The quad-copter firmware was compiled for ARM32, EABI5 version 1, and statically linked. We targeted four versions of the quad-copter's stabilization algorithm, a compiled version present in a firmware image on Github [109], a version compiled from the specification given by the algorithm's original publication [108], a *correct* version of the algorithm written for Matlab, and a *correct* version of the algorithm written in C, compiled using the same ARM compiler, and then lifted by InteGreat.

Table 6.1: 14 of the 18 quad-copter *binds* array elements used to lift continuous equations from binary firmware using InteGreat.

Program slice C code name	**Associated *binds* array**
__addsf3	"o0": "binds['i0'] + binds['i1']"
__aeabi_f2d	"tmp": "z3.fpToFP(z3.RNE(), binds['i0'], z3.FPSort(11, 53))", "o0": "z3.Extract(31, 0, z3.fpToIEEEBV(binds['tmp']))", "o1": "z3.Extract(63, 32, z3.fpToIEEEBV(binds['tmp']))"
__aeabi_fcmpeq	"o0": "z3.If(binds['i0'] == binds['i1'], z3.FPVal(1.0, z3.FPSort(8, 24)), z3.FPVal(0.0, z3.FPSort(8, 24)))"
__aeabi_fdiv	"o0": "binds['i0'] / binds['i1']"
__aeabi_fmul	"o0": "binds['i0'] * binds['i1']"
__aeabi_fsub	"o0": "binds['i0'] - binds['i1']"
__divdf3	"tmp": "z3.fpBVToFP(z3.Concat(binds['i1'], binds['i0']), z3.FPSort(11, 53)) / z3.fpBVToFP(z3.Concat(binds['i3'], binds['i2']), z3.FPSort(11, 53))", "o0": "z3.Extract(31, 0, z3.fpToIEEEBV(binds['tmp']))", "o1": "z3.Extract(63, 32, z3.fpToIEEEBV(binds['tmp']))"
__muldf3	"tmp": "z3.fpBVToFP(z3.Concat(binds['i1'], binds['i0']), z3.FPSort(11, 53)) * z3.fpBVToFP(z3.Concat(binds['i3'], binds['i2']), z3.FPSort(11, 53))", "o0": "z3.Extract(31, 0, z3.fpToIEEEBV(binds['tmp']))", "o1": "z3.Extract(63, 32, z3.fpToIEEEBV(binds['tmp']))"
__truncdfsf2	"o0": "z3.fpToFP(z3.RNE(), z3.fpBVToFP(z3.Concat(binds['i1'], binds['i0']), z3.FPSort(11, 53)), z3.FPSort(8, 24))"
asin	"tmp": "z3.Function('asin', *[z3.BitVecSort(32) for _ in range(2)], z3.BitVecSort(64))(binds['i0'], binds['i1'])", "o0": "z3.Extract(31, 0, binds['tmp'])", "o1": "z3.Extract(63, 32, binds['tmp'])"
atan2	"o0": "z3.Extract(31, 0, z3.Function('atan2', *[z3.BitVecSort(32) for _ in range(4)], z3.BitVecSort(64))(binds['i0'], binds['i1'], binds['i2'], binds['i3']))", "o1": "z3.Extract(63, 32, z3.Function('atan2', *[z3.BitVecSort(32) for _ in range(4)], z3.BitVecSort(64))(binds['i0'], binds['i1'], binds['i2'], binds['i3']))"
sqrt	"tmp": "z3.fpSqrt(z3.RNE(), z3.fpBVToFP(z3.Concat(binds['i1'], binds['i0']), z3.FPSort(11, 53)))", "o0": "z3.Extract(31, 0, z3.fpToIEEEBV(binds['tmp']))", "o1": "z3.Extract(63, 32, z3.fpToIEEEBV(binds['tmp']))"
neg	"o0": "-binds['i0']"
csqrt	"o0": "z3.Function('csqrt', z3.FPSort(8, 24), z3.FPSort(8, 24))(binds['__aeabi_f2d']['i0'])"

Function	#Lifts	#Symbols	#LoA
integral	0	5	47
derivative	0	10	24
pid	8	9	302
main	2	6	148

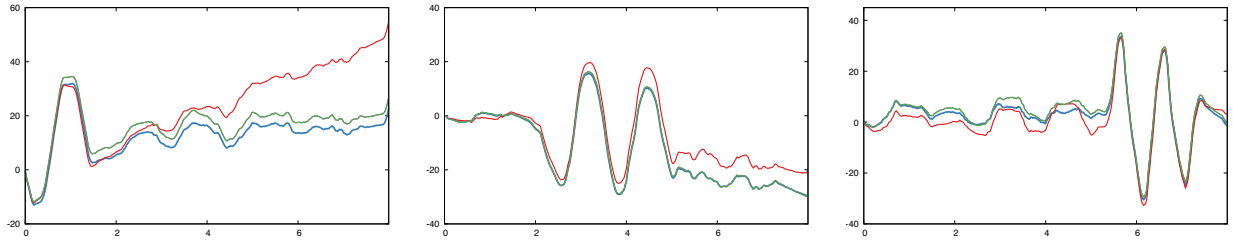
Figure 6.2: Analytics for InteGreat's lifting of the PLC firmware's control logic. The integral and derivative functions contained no inner program slices. Each call to PID, integral, and derivative maintains separate state.

The inputs to InteGreat used to lift the quad-copter firmware consisted of 18 program slices (γ) and 24 *binds* entries (Φ). We depict the results of these lifting rules in Fig. 6.1, and include the specification of 16 these bindings in Table 6.1 as an example. The left side demonstrates the structure by which we were able to nest lifting rules. The right side demonstrates the modularity of lifting rule applications across different code locations, i.e. the slice for lifting a floating point multiply could be applied across several distinct code locations.

The PLC firmware was compiled both using CoDeSys 2.3.9.44 and e!COCKPIT 3.5.16.30 for the WAGO 750-881 and PFC200 G2 750-8217 PLCs, respectively. We recovered the same results for both of these PLCs and present the results for the latter in this paper, as it is the more recent version. Both PLCs use the same CPU architecture (ARM32), and the firmware is statically linked.

We targeted an algorithm in PLC firmware for actuating a pressure-controlling valve in the Tennessee Eastman Challenge [126], a benchmark for modeling and analyzing the dynamics of industrial control systems. The firmware is written in the structured text, IEC 61131-3, programming language and then compiled to a binary for the PLC. The firmware in question was provided by ICSREF [110], a security paper which staged a code upload attack on a physical PLC connected to a Matlab environmental model. Additional details of this attack's physical setup are given in [127].

The specification used to lift the PLC firmware consisted of 3 program slices and 3 *binds* entries generating uninterpreted functions for three cases: a proportional-integral-derivative call, a derivative call, and an integral call. We depict the results of these lifting rules in Fig. 6.2. The PLC had significantly fewer binds as it did not rely on a software floating point ABI and Matlab, the target domain, had pre-existing functions for integral and derivative.



(a) Pitch (b) Yaw (c) Roll

Figure 6.3: Empirical results measuring the ability of InteGreat's recovered models of Orientation Estimation algorithms to identify bugs.

6.4.1 Correct Lifting of a Quad-Copter Stabilization Algorithm

We evaluated InteGreat by recovering equations for continuous orientation estimation from an open-source, MCU-based quad-copter autopilot controller. The firmware we studied implemented Madgwick's Gradient Descent Orientation Filter [108]. The orientation estimator takes in sensor inputs from the IMU consisting of an accelerometer, gyroscope, and magnetometer. Orientation Estimators for strapdown INS (Inertial Navigation Systems) *continuously* fuse sensor data with previous estimations and obtain new attitude across the time domain. Because of this, difference in implementations that are subtle on the surface lead to large accumulative error across small time periods.

We ran four experiments using different versions of Madgwick's algorithm to showcase InteGreat's ability in detecting bugs and algorithm variants in firmware. These experiments are depicted in Fig. 6.3, through a simulation of the quad-copter spinning in three dimensions. These simulations were performed using Matlab's "rpy_9axis" sensor data.

Note that only three lines are present in Fig. 6.3. The fourth line, the correct Matlab implementation, the blue line, is identical to lifted version of a corresponding correct C implementation. This finding ensures InteGreat did not make mistakes while lifting, and was able to produce identically behaving Matlab code with respect to compiled C firmware code. The equivalence between the two correct implementations gives us confidence in the methodology InteGreat introduces.

The figure depicts the same inputs to the IMU sensor of a device oscillating in pitch, yaw, then roll. **Green** is lifted by InteGreat from *real world* firmware that contains a bug. **Blue** contains *two* lines: the InteGreat lifted firmware recompiled to correct the gradient calculation, and *a correct* Matlab implementation of Madgwick's algorithm. **Red**, however, is a compiled firmware using the *faulty* C code included in Madgwick's published report. This version has accumulative error with respect to the earth's magnetic field.

Continuous equations lifted by InteGreat were identical to ground truth simulations in a controlled IMU sensor experiment rotating the quadcopter in 3 dimensions using 1,600 measurement points. This demonstrates our framework is able to lift correct continuous equations from discrete, machine code implementations and verify them using tools operating over the abstract domain.

Example of InteGreat Identifying Bugs We next evaluate whether InteGreat is useful when attempting to identify bugs in firmware. When comparing the Madgwick-provided Matlab implementation (blue) with the version recovered from the *real world* firmware implementation (green), we discovered an error in the gradient of the solution surface which is calculated by multiplying the objective Function matrix and its Jacobian matrix: $\nabla F = \mathbf{J}^T({}^S_E\hat{\mathbf{q}}, {}^E\hat{\mathbf{b}})\mathbf{F}({}^S_E\hat{\mathbf{q}}, {}^E\hat{\mathbf{b}}, {}^S\hat{\mathbf{a}}, {}^S\hat{\mathbf{m}})$. Comparing the correct magnetic field components of the Objective Function with the InteGreat lifted version (using normalized magnetic field sensor values for simplicity):

$$\mathbf{f}_b({}^S_E\hat{\mathbf{q}}, {}^E\hat{\mathbf{b}}\,{}^S\hat{\mathbf{m}}) = \begin{bmatrix} 2b_x(0.5 - q_3^2 - q_4^2) + 2b_z(q_2q_4 - q_1q_3) - m_x \\ 2b_x(q_2q_3 - q_1q_4) + 2b_z(q_1q_2 + q_3q_4) - m_y \\ 2b_x(q_1q_3 + q_2q_4) + 2b_z(0.5 - q_2^2 - q_3^2) - m_z \end{bmatrix}$$

$$F_4 = ((0.5 + (-q_3^2) + (-q_4^2)) * b_x + (q_2 * q_4 + (-q_1 * q_3)) * b_z + (-m_x)$$
$$F_5 = (q_2 * q_3 + (-q_1 * q_4)) * b_x + (q_3 * q_4 + q_1 * q_2) * b_z + (-m_y)$$
$$F_6 = (((0.5 + (-q_2^2)) + (-q_3^2)) * b_z + (q_1 * q_3 + q_2 * q_4) * b_x + (-mz)$$

It is clear that terms using the earth's magnetic field (${}^E\hat{\mathbf{b}}$) in the Objective Function have missing coefficients.[1] The same errors occur in all magnetic field components of the Jacobian. We compare only one row of the Jacobian here for simplicity:

$$J_{4,1} = (-q_3 * b_z)$$
$$J_{4,2} = (q_4 * b_z)$$
$$J_{4,3} = (q_3 * (-2 * b_x) + (-q_1 * b_z))$$
$$J_{4,4} = (q_4 * (-2 * b_x) + (q_2 * b_z))$$

$$\mathbf{J_4}({}^S_E\hat{\mathbf{q}}, {}^E\hat{\mathbf{b}}) = \begin{bmatrix} -2b_zq_3, & 2b_zq_4, & -4b_xq_3 - 2b_zq_1, & -4b_xq_4 + 2b_zq_2 \end{bmatrix}$$

After correcting all coefficients of the earth's magnetic field (${}^E\hat{\mathbf{b}}$) and recompiling

[1] b_y is not present as the earth's magnetic field is be considered to have components in one horizontal axis and the vertical axis.

the quad-copter firmware, the continuous equations lifted by InteGreat exactly matched Madgwick's Matlab implementation (blue).

We then compared the corrected equations (blue) with a *third* firmware version, compiled to use Madgwick's published implementation, written in C, which is included in the original Madgwick paper (red). We found that the published variant of the algorithm recursively feeds in previously calculated gyroscopic biases multiplied by a correction constant ζ. In contrast, existing implemented algorithms did *not* perform gyroscopic bias removal and instead used dynamically calculated flux vectors of the Earth's frame in the equation's objective function. This further confirmed InteGreat's ability to help researchers understand differences between implementations and published results.

By lifting compiled machine code to continuous equations using logical reasoning, it is possible to identify bugs in firmware implementations and check the validity of published results. Validation can be performed by comparing representations which, after lifting, operate over the same domain, or by using a framework to check output equivalence, i.e. reachability, between algorithm versions.

6.4.2 Analyzing and Extracting Functions from PLC Firmware

In this section, we demonstrate our ability to analyze continuous equation models of the firmware of a PLC (Fig. 6.2). We recovered the following core model from the firmware's PID (proportional-integral-derivative) functions, where I and D are uninterpreted functions for integral and derivative, respectively:

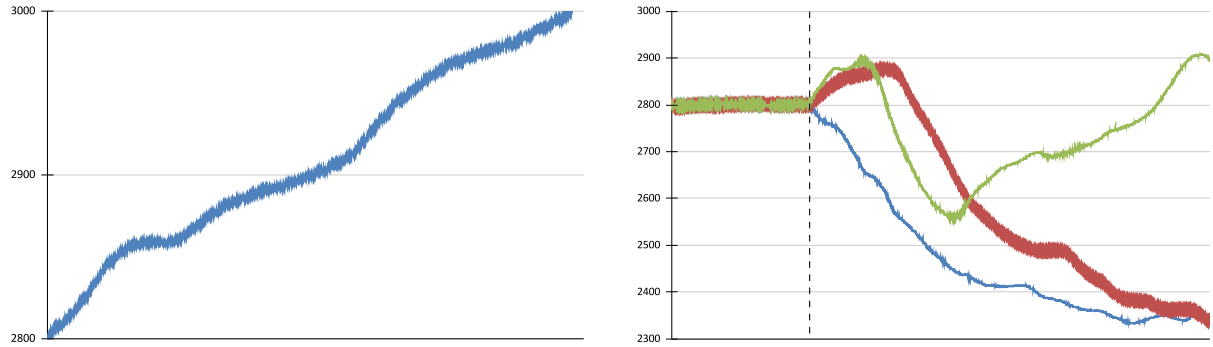
$$v_6((v_3 - v_2) + \frac{1}{v_5} I(v_3 - v_2, 1000t) + v_1 D(v_3 - v_2, 1000t)) + v_0 + v_7 \tag{6.1}$$

For analysis in Matlab, we needed to also take into account some non-infinitesimal measure of time. We simplified the recovered equation further via a few additional common axioms, e.g. $new\_sym = v_3 - v_2$, $integ\_e\_dt = I(binds_{i0}, binds_{i1})$:

$$K_p(e + \frac{1}{\tau_i} \int e\mathrm{d}t + \tau_d dvet) + b \tag{6.2}$$

Eqn. 6.2 more closely matches the traditional PID formulation, where K_p is the proportional gain constant the ICSREF attack modified. The other equations InteGreat recovered from the PLC did not require further constructed abstraction specifications.

Discovering Additional Hardware. In the lifted version of the firmware, we also found an obscure scaling was applied to the sensor value for the input reactor pressure before



(a) Naive Attack Reproduction

(b) Voltage Corrected Destabilization

Figure 6.4: **Left**: An attempt to stage the ICSREF attack using the recovered model without taking into account the physical effects of digital-to-analog conversion. The plant shuts down upon when the reactor pressure hits 3000 kPa. **Right**: Variations of the staged attack with physical effects accounted for. The dashed line is the point at which the attack occurs (the 4 hour mark).

it was supplied to Eqn. 6.2:

$$P = 1000(((P_{digital}/30000) - 0.0046)/0.9876) + 2000 \tag{6.3}$$

We also were recovering the wrong value for the reactor pressure when we staged normal operation of the environmental model using our lifted equations, depicted in Fig. 6.4a. This led us to discover the reactor pressure (which starts at 2800 kPa in the TE simulation) was supplied to the PLC through a *physical wire* connected to a separate Analog-to-Digital Converter (ADC), rather than a serial port. The original pressure value is converted to a voltage value between 1 and 3, and then this voltage is converted back to a digital value on the physical PLC (30,000) corresponding to 3 volts.

We confirmed this was the case with the ICSREF authors. Because of the the physical effects of the ADC being taken into account, the scaling equations applied to PLC I/O values in the Matlab environmental model used by the paper were *not* the inverse of Eqn. 6.3. Instead, the Matlab environmental model had no scaling to account for the ADC and the firmware *implicitly* suggested the ADC was present via the scaling computation.

Even without ground truth, we were able to discover the presence of a physical ADC by analyzing a firmware's equations after lifting. This demonstrates novel discoveries about a system's environment can be made by using InteGreat to analyze firmware.

6.5 PRECISE DESTABILIZATION ATTACK DEVELOPMENT

The ICSREF attack flipped the sign on the proportional gain constant of the first of the firmware's PID calls. By recovering the continuous equations the firmware image implemented, we were able to *stage* this attack without access to the physical PLC.

A naive extraction of the firmware's implementation into Matlab, maintaining the original voltage conversions discussed above, results in Fig. 6.4a when staging the attack. This is because the input and output values to the PLC's equations are affected by the scaling decided by the physical digital-to-analog and analog-to-digital converters. This proposes the importance of a hybrid approach to modeling in the current domain of firmware rehosting [128, 129, 130]. While an exact emulation of the firmware's operation is desirable for dynamic analysis and testing (e.g. fuzzing), it is also necessary to verify that the I/O boundaries of the system are consistent with the physical environment or environmental model the emulation is attached to.

Moving forward with this understanding, we were then able to correctly reproduce the destabilizing effects on reactor pressure created by uploading code to the PLC (Fig. 6.4b). **Red** represents the original PLC firmware behavior from the ICSREF paper. However, notice that the paper included an unexplained positive bump at the point of attack. Lifted equations allow us to explore an idealized model of the firmware's implementation: by removing all scaling operations from Matlab and the recovered model, we attained the line in **Blue**. In doing so, we discovered exactly how much the ADC voltage conversion affects the cyberphysical system's dynamics. The **Green** line represents a case where the attack *did not* occur but control over the reactor pressure is given to the PLC (with ADC scaling). This scaling makes the pressure of the plant far less stable, a feature of the system not addressed in the original ICSREF paper.

As the above evaluation demonstrates, using InteGreat we were able to both reproduce and verify the existing *ICSREF* attack on a PLC and extend it to target an arbitrary reactor pressure levels by modifying the proportional gain constant in the recovered model to desired values. The recovered equations were of value in *precisely* destabilizing the plant's reactor pressure while maintaining the appearance of correct behavior, similar to stuxnet [131]. Because the recovered models of the PLC's control equations are general, it was also possible to experiment with any variety of potential modifications to the PID controller and explore a variety of reachable states given the environmental model.

6.6 DISCUSSION

Informed by both the Jetset and Edact-Ray works, InteGreat adopts an opinionated perspective on the challenges facing binary program analysis. It attempts to address universal and necessary limitations to abstract interpretation, such as the modeling of microarchitectural semantics and state explosion, by applying the well-worn technique of wrapping these complexities in a layer of indirection, and then automates the construction of this indirection. While this alleviates the *immediate* difficulties involved in inferring loop invariants and pointer analysis, InteGreat also has the explicit limitation of requiring users to leave these semantics undefined or provide their own formulae for resolution.

We therefore consider a primary future application of InteGreat being *lifting libraries*. By incorporating an object-oriented approach for abstraction specifications, the InteGreat framework is able to continually expand its base of knowledge. A given set of lifting rules can be written and then shared, similar to a library, for the analysis of wide ranges of firmware binaries. These libraries may also be generated via automated methods, and serve as a compressed encoding of more complex inference procedures. Moreover, because it is possible to encode uniform semantics for a given system, these libraries can begin to function as "contracts" to ensure a system meets certain regulatory implementation requirements, something top-down verification cannot provide.

InteGreat does not solve the *internal* or *external* semantic problems posed in Section 4.4 or the more general problem of missing information in a system's specification. Instead, the system attempts to provide a better interface for management of this problems. It does so by making no assumptions about the underlying semantics of the system, building a model of the system from the ground up by starting with the implementation, rather than a mathematical model.

In providing an interface for a more abstract representation the system during symbolic execution, InteGreat is able to rely on additional input for undefined or undecidable cases and extract useful models in cases where no such additional information is required or feasible to provide. InteGreat thereby alleviates the need to treat complex systems as complete black-boxes,[2] and instead attempts to restrict the use of black-boxes to cases where information on the system is legitimately missing.

[2]The dangers of making representational assumptions were demonstrated in Chapter 5.

6.7 SUMMARY

The work presented in this chapter empirically demonstrates that traditional top-down approaches to verification can underestimate the complexity of firmware implementations. Instead of *only* working from mathematical models of a specific implemented system, we propose the additional application of careful, logic-based function extraction. From this point of view, it is possible to define the system at arbitrary levels of granularity while ensuring that the provided representation for verification is accurate to the system under study. For this purpose, we developed a system that applies logic-based function extraction to recover continuous equations from symbol-stripped binary firmware. Through the analysis of two real world systems, a quad-copter and PLC, we were able to demonstrate such an approach's ability to find differences between published, high-level representations and low-level system representations. We were also able to elucidate otherwise unknowable features of a system under study, such as physical ADCs and the effects of their value representation on the system's dynamics.

Chapter 7: Conclusions

Analysis of complex systems with limited ground-truth information is difficult but critical. Existing approaches proceed either from the minutia of a system's microarchitectural semantics or a broad set of general properties which are not always applicable to a system's implementation specifics and attempt to meet the other side in the middle. However, we are still facing profound challenges in explicating the problems involved in the verification of systems, and often place them under the notion of undecidability.

In this dissertation, we examined the practice of emulation and simulation of systems, and developed three novel applications of emulation to systems security measurement. Our first analysis considered the problem of modeling hardware from the domain of firmware. Second, we considered inferring text removed from a PDF document by examining residual information and universally quantifying over possible inputs to a simulation of the document's production software. Last, we provided a framework for the extraction of otherwise black-box system functions using a combination of formal reasoning and abstract interpretation. All of these techniques identified new information or perspectives on the system in consideration, and then used this novel insight to infer missing information. Emulation always provides uncertain bounds on the true behavior of the system modeled, defined by the method in which the system is emulated. Each of our different approaches each eventually converged to some bound on the knowable, and we identified how this bound plays a role when considering the correctness of deductions made from the model. Such considerations can be understood in two ways. In reference to missing subcomponents, e.g. redacted content and missing peripherals, the unknown can be used to bound our willingness to consider a system secure: referring to the absolute rather than simple opacity, what cannot be effectively disambiguated to a necessary degree is meaningless to an adversary, and that which can be known is insecure. In the development of emulations, this bound informed the construction of a representation for the system's behavior, and directed us towards completion in our analysis of what can be observed, i.e. the shifting schemes underlying text layout and the representations used for reasoning about systems.

We introduced different strategies to model the dynamics of diverse complex systems where ground-truth information was limited or unavailable. In Chapter 4, we used taint-tracking based symbolic execution, fuzzing, and live binary rewriting to find exploits for a critical avionics component. Because these techniques work at the level of microarchitectural semantics, we were able to generalize each of these principles to the measurement

of security of many embedded systems. Our findings ultimately led to the deeper understanding that new strategies were needed for reasoning about complex relationships within known firmware information and identifying unknown external components. In Chapter 5, we approached the latter of these challenges by reviving a decades-old information theoretic approach to the unknown: dictionary attacks, and effectively applied this technique to achieve groundbreaking results that rendered hundreds of redactions in historically relevant documents ineffective. The tool we built for this purpose, Edact-Ray, adapted the careful analysis of binary code introduced by Jetset in the previous chapter to the generation of precise simulations of PDF production, allowing us to use the information in the PDF documents as an "oracle" to eliminate bad guesses for redacted content. In Chapter 6, we approached the problem of function extraction faced by Jetset and Edact-Ray, in order to present a bottom-up approach to cyber-physical system verification, bootstrapping mathematical models for systems from symbol-stripped binary code. We were able to use this system to identify differences between published mathematical models and their implemented versions, as well as recover control equations for a PLC used in a chemical plant, then perform more precise analysis and attack modeling against these equations than was present in original publications on the system.

Each of these techniques provided impactful analysis of otherwise opaque software and embedded systems, in part due to the specific efforts involved in developing the tools used. They present a clear line of development in approaching the problem of security measurement in critical systems with specific applications and areas of focus, and each of these areas could be further explored to identify new explots. There are also many additional challenges remaining in the field of emulation construction. We provide an overview of potential areas for future work:

Multi-System Interaction Emulation and Simulation Extraction An immediate challenge to our analysis of PDF document production schemes in Chapter 5 was the possiblity of modeling and tracking information as it flowed through multiple different software stacks. For example, a PDF produced by Microsoft Word can then be fed to Apple's email client, where it is then opened in firefox and printed as a PDF using the system dialog. Exactly identifying the operations that are performed on the PDF's specification in each of these steps requires recording each step's execution individually, reverse engineering each software stack individually, or instrumenting the operating system to track how the information of interest in propogated between these systems. One avenue for future work is to instrument a system like QEMU, or the operating system itself, so that precise extraction of models of operations performed on data can be extracted and reused in

simulations for the purposes of, for example, the measurement of redaction security.

Type Checking for Program Analysis and Emulation The representations used to emulate, represent, and verify the behavior of systems are usually partial due to the complexity of reproducing and precisely understanding the system under test. InteGreat's approach to function extraction introduced a verifiable representation of semantic modeling: we can represent the relationships and translations between symbols logically. This effectively provides a form of type system for the interpetation of the analyzed system. It would be ideal to extend this type system with analyses built from a description of a system to ensure symbolic translations accurately represent the concrete system and do not remove or fail to account for critical components that could affect the correctness of further analysis, e.g. program analysis built on top of QEMU can simply ignore some TCG helper routines which are used by the analyzed binary and the type checker would enforce that these are supported. This could also help to ensure the authenticity of academic research in the subject of security.

Catalogs of Common System Functions Use of the Jetset system in Chapter 4 was severely limited by the unsolvable problem of understanding the "correct" path to explore through an arbitrary firmware image. The simplest and most effective solution to this problem and many of the challenges encountered by this dissertation is the effective cataloging, representation, and sharing of semantic knowledge regarding the function of systems. For example, when performing symbolic execution on a function, the chances are that the same function or one quite similar to it has been previously symbolically executed by some individual analyzing some system at some point in time. Therefore, it would be valuable to accumulate a database of both system functions and associated strategies for their interpretation, which could be queried by analysis routines like abstract interpretation. Moreover, this could provide predicates for pattern matching that would be able to quickly determine whether a given subfunction of a new system was secure or insecure based upon the history of that function's use. In each case, the specific representation of the information may vary, but there are a range of existing security analyses are uniform enough in behavior, such as fuzzing, that the development of such a catalogs is possible and in some cases already underway.

References

[1] " Published CVE Records Metrics ," https://www.cve.org/About/Metrics, Mitre, 2022.

[2] J. Pantiuchina, M. Lanza, and G. Bavota, "Improving Code: The (Mis)Perception of Quality Metrics," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 80–91.

[3] J. Feigenspan, S. Apel, J. Liebig, and C. Kastner, "Exploring Software Measures to Assess Program Comprehension," in *2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2011, pp. 127–136.

[4] D. G. Feitelson, "From Code Complexity Metrics to Program Comprehension," *Communications of the ACM*, vol. 66, no. 5, pp. 52–61, 2023.

[5] S. Stolfo, S. M. Bellovin, and D. Evans, "Measuring Security," *IEEE Security & Privacy*, vol. 9, no. 3, pp. 60–65, 2011.

[6] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel et al., "Sok:(State of) the Art of War: Offensive Techniques in Binary Analysis," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 138–157.

[7] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck, "Dos and Don'ts of Machine Learning in Computer Security," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3971–3988.

[8] Z. Chen, S. L. Thomas, and F. D. Garcia, "MetaEmu: An Architecture Agnostic Rehosting Framework for Automotive Firmware," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 515–529.

[9] E. Johnson, M. Bland, Y. Zhu, J. Mason, S. Checkoway, S. Savage, and K. Levchenko, "Jetset: Targeted Firmware Rehosting for Embedded Systems," in *USENIX Security Symposium*, 2021, pp. 321–338.

[10] M. Bland, A. Iyer, and K. Levchenko, "Story Beyond the Eye: Glyph Positions Break PDF Text Redaction," *Privacy Enhancing Technologies Symposium*, 2023.

[11] M. Bland, A. Chen, and K. Levchenko, "InteGreat: Lifting Continuous Control Equations from Binary Code," *IEEE/ACM International Conference on Automated Software Engineering (in Submission)*, 2023.

[12] P. Marwedel, *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. Springer Nature, 2021.

[13] T. Ball, "The Concept of Dynamic Analysis," *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6, pp. 216–234, 1999.

[14] B. Chess and G. McGraw, "Static Analysis for Security," *IEEE security & privacy*, vol. 2, no. 6, pp. 76–79, 2004.

[15] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal Methods: Practice and Experience," *ACM computing surveys (CSUR)*, vol. 41, no. 4, pp. 1–36, 2009.

[16] G. Leeb and N. Lynch, "Proving Safety Properties of the Steam Boiler Controller: Formal Methods for Industrial Applications: A Case Study," *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, pp. 318–338, 2005.

[17] S.-K. Huang, M.-H. Huang, P.-Y. Huang, H.-L. Lu, and C.-W. Lai, "Software Crash Analysis for Automatic Exploit Generation on Binary Programs," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 270–289, 2014.

[18] Y. Le Corre, J. Großschädl, and D. Dinu, "Micro-Architectural Power Simulator for Leakage Assessment of Cryptographic Software on ARM Cortex-M3 Processors," in *Constructive Side-Channel Analysis and Secure Design: 9th International Workshop, COSADE 2018, Singapore, April 23–24, 2018, Proceedings 9*. Springer, 2018, pp. 82–98.

[19] O. Landsiedel, K. Wehrle, and S. Gotz, "Accurate Prediction of Power Consumption in Sensor Networks," in *The Second IEEE Workshop on Embedded Networked Sensors, 2005. EmNetS-II.* IEEE, 2005, pp. 37–44.

[20] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, "Challenges in Firmware Re-Hosting, Emulation, and Analysis," *ACM Computing Surveys (CSUR)*, vol. 54, no. 1, pp. 1–36, 2021.

[21] S. Delaune and F. Jacquemard, "A Theory of Dictionary Attacks and Its Complexity," in *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.* IEEE, 2004, pp. 2–15.

[22] T. Reps, G. Balakrishnan, and J. Lim, "Intermediate-Representation Recovery from Low-Level Code," in *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, 2006, pp. 100–111.

[23] B. Feng, A. Mera, and L. Lu, "P^{2}IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling," in *Proceedings of USENIX Security 2020*, Aug. 2020.

[24] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation," in *Proceedings of USENIX Security 2020*, Aug. 2020.

[25] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a Survey," *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.

[26] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation," in *USENIX Security Symposium*, 2019, pp. 1099–1114.

[27] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "{QSYM}: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 745–761.

[28] L. Borzacchiello, E. Coppa, and C. Demetrescu, "FUZZOLIC: Mixing Fuzzing and Concolic Execution," *Computers & Security*, vol. 108, p. 102368, 2021.

[29] "SEL-751A Feeder Protection Relay," https://selinc.com/products/751A/, Schweitzer Engineering Laboratories, 2023.

[30] "CMU-900 Communication Management Unit," https://www.collinsaerospace.com/what-we-do/industries/commercial-aviation/flight-deck/communications/data-link-services/cmu-900-communication-management-unit, Collins Aerospace, 2023.

[31] "Raspberry Pi," https://www.raspberrypi.com/, Raspberry Pi Foundation, 2023.

[32] "BeagleBoard-xM," https://beagleboard.org/beagleboard-xm, beagleboard.org, 2023.

[33] W. Zhou, L. Zhang, L. Guan, P. Liu, and Y. Zhang, "What Your Firmware Tells You Is Not How You Should Emulate It: A Specification-Guided Approach for Firmware Emulation," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 3269–3283.

[34] G. Hernandez, M. Muench, D. Maier, A. Milburn, S. Park, T. Scharnowski, T. Tucker, P. Traynor, and K. Butler, "FIRMWIRE: Transparent dynamic analysis for cellular baseband firmware," in *Network and Distributed Systems Security Symposium (NDSS) 2022*, 2022.

[35] Y. Sun, Z. Li, S. Lv, and L. Sun, "Spenny: Extensive ICS Protocol Reverse Analysis via Field Guided Symbolic Execution," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[36] "Advanced RECAP Search," https://www.courtlistener.com/recap/, Court Listener, 2023.

[37] "Digital National Security Archive," https://nsarchive.gwu.edu/digital-national-security-archive, George Washington University, 2021.

[38] "Public Access to Court Electronic Records," https://pacer.uscourts.gov/, Administrative Office of the U.S. Courts, 2021.

[39] "All Inspector General Reports in one Place," Council of Inspectors General on Integrity and Efficiency, 2021.

[40] "The Government Attic," governmentattic.org, 2021.

[41] M. Burgess, "Redacted Documents Are Not as Secure as You Think," https://www.wired.com/story/redact-pdf-online-privacy/, Wired, 2022.

[42] "Simulink," https://www.mathworks.com/products/simulink.html, MathWorks, 2023.

[43] 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.

[44] D. Quinlan, "ROSE: Compiler Support for Object-Oriented Frameworks," *Parallel processing letters*, vol. 10, no. 02n03, pp. 215–226, 2000.

[45] C. Eagle and K. Nance, *The Ghidra Book: The Definitive Guide*. no starch press, 2020.

[46] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha, "Testing Intermediate Representations for Binary Analysis," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 353–364.

[47] J. Stanier and D. Watson, "Intermediate Representations in Imperative Compilers: A Survey," *ACM Computing Surveys (CSUR)*, vol. 45, no. 3, pp. 1–27, 2013.

[48] A. Letichevsky, O. Letychevskyi, V. Skobelev, and V. Volkov, "Cyber-Physical Systems," *Cybernetics and Systems Analysis*, vol. 53, pp. 821–834, 2017.

[49] M. Bland, "CMU-900 Local Code Execution and Rootkit," https://youtu.be/r4M9AFZcj2w, 2022.

[50] E. Johnson and M. Bland, "Jetset," https://github.com/aerosec/jetset, 2021.

[51] M. Bland, "Deredaction," https://github.com/maxwell-bland/deredaction, 2023.

[52] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX annual technical conference, FREENIX Track*, vol. 41, 2005, p. 46.

[53] N. A. Quynh and D. H. Vu, "Unicorn: Next generation cpu emulator framework," *BlackHat USA*, vol. 476, 2015.

[54] Z. Deng, X. Zhang, and D. Xu, "Bistro: Binary Component Extraction and Embedding for Software Security Applications," in *Computer Security–ESORICS 2013: 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings 18*. Springer, 2013, pp. 200–218.

[55] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary Code Extraction and Interface Identification for Security Applications," California Univ. Berkeley Dept. of Electrical Engineering and Computer Science, Tech. Rep., 2009.

[56] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 488–498.

[57] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti et al., "AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares," in *NDSS*, vol. 23, 2014, pp. 1–16.

[58] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *OSDI*, vol. 8, 2008, pp. 209–224.

[59] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte et al., "ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–31, 2019.

[60] *Arm Architecture Reference Manual for A-Profile Architecture*. ARM, 2023.

[61] "A deep dive into QEMU: The Tiny Code Generator (TCG), part 1," https://airbus-seclab.github.io/qemu_blog/tcg_p1.html, Airbus Seclab, 2023.

[62] M. Zalewski, "Technical "Whitepaper" for afl-fuzz," http://lcamtuf.coredump.cx/afl/technical_details.txt, 2017.

[63] O. Sery, G. Fedyukovich, and N. Sharygina, "Interpolation-Based Function Summaries in Bounded Model Checking," in *Proceedings of the 7th International Haifa Verification Conference on Hardware and Software: Verification and Testing*, ser. HVC'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 160–175.

[64] L. Alt, S. Asadi, H. Chockler, K. Even Mendoza, G. Fedyukovich, A. E. Hyvärinen, and N. Sharygina, "HiFrog: SMT-based Function Summarization for Software Verification," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2017, pp. 207–213.

[65] S. Asadi, M. Blicha, A. Hyvärinen, G. Fedyukovich, and N. Sharygina, "Incremental Verification by SMT-based Summary Repair," in *2020 Formal Methods in Computer Aided Design (FMCAD)*, vol. 1. TU Wien Academic Press, 2020, pp. 77–82.

[66] T. A. Henzinger, "The Theory of Hybrid Automata," in *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 1996, pp. 278–292.

[67] C. E. Shannon, "A Mathematical Theory of Communication," *The Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[68] A. Sabelfeld and A. C. Myers, "Language-Based Information-Flow Security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[69] G. Lowe, "Quantifying Information Flow," in *Proceedings 15th IEEE Computer Security Foundations Workshop. CSFW-15*. IEEE, 2002, pp. 18–31.

[70] J. H. Van Lint, *Coding Theory*. Springer, 1971, vol. 201.

[71] T. Richardson and R. Urbanke, *Modern Coding Theory*. Cambridge university press, 2008.

[72] J. Bierbrauer, *Introduction to Coding Theory*. CRC Press, 2016.

[73] T. M. Cover, *Elements of Information Theory*. John Wiley & Sons, 1999.

[74] "Time Travel Debugging - Overview," https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/time-travel-debugging-overview, Microsoft, 2023.

[75] M. Campion, M. Dalla Preda, and R. Giacobazzi, "Partial (In)completeness in Abstract Interpretation: Limiting the Imprecision in Program Analysis," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–31, 2022.

[76] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 337–340.

[77] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices." in *NDSS*, 2018.

[78] D. D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware," in *Proceedings of NDSS 2016*, Feb. 2016.

[79] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna, "Toward the Analysis of Embedded Firmware through Automated Re-hosting," in *Proceedings of RAID 2019*, Sep. 2019, pp. 135–150.

[80] M. J. Renzelmann, A. Kadav, and M. M. Swift, "SymDrive: Testing drivers without devices," in *Proceedings of OSDI 2012*, Oct. 2012, pp. 276–292.

[81] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution," in *Proceedings of USENIX Security 2013*, Aug. 2013, pp. 463–478.

[82] V. Chipounov and G. Candea, "Reverse engineering of binary device drivers with RevNIC," in *Proceedings of EuroSys 2010*, Apr. 2010, pp. 167–180.

[83] G. Hernandez, F. Fowze, D. J. Tian, T. Yavuz, and K. Butler, "FirmUSB: Vetting USB device firmware using domain informed symbolic execution," in *Proceedings of CCS 2017*, Oct. 2017.

[84] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares," in *Proceedings of NDSS 2014*, Feb. 2014.

[85] K. Koscher, T. Kohno, and D. Molnar, "SURROGATES: Enabling near-real-time dynamic analyses of embedded systems," in *Proceedings of WOOT 2015*, Aug. 2015.

[86] M. Kammerstetter, C. Platzer, and W. Kastner, "Prospect: peripheral proxying supported embedded code testing," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014, pp. 329–340.

[87] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A Large-Scale Analysis of the Security of Embedded Firmwares," in *Proceedings of USENIX Security 2014*, Aug. 2014, pp. 95–110.

[88] A. Costin, A. Zarras, and A. Francillon, "Automated Dynamic Firmware Analysis at Scale: a Case Study on Embedded Web Interfaces," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 437–448.

[89] K.-K. Ma, K. Yit Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Proceedings of SAS 2011*, Sep. 2011, pp. 95–111.

[90] *BeagleBoard-xM Rev C System Reference Manual*, http://beagleboard.org/static/BBxMSRM_latest.pdf, BeagleBoard.org Foundation, Apr. 2010, revision 1.0.

[91] "Docker," docker.com, Docker, Inc, 2020.

[92] R. S. Venkatesh, T. Smejkal, D. S. Milojicic, and A. Gavrilovska, "Fast In-Memory CRIU for Docker Containers," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 53–65.

[93] S. Crow, B. Farinholt, B. Johannesmeyer, K. Koscher, S. Checkoway, S. Savage, A. Schulman, A. C. Snoeren, and K. Levchenko, "Triton: A Software-Reconfigurable Federated Avionics Testbed," in *Proceedings of CSET 2019*, 2019.

[94] H. Dixon, "Embarrassing Redaction Failures," https://www.americanbar.org/groups/judicial/publications/judges_journal/2019/spring/embarrassing-redaction-failures/, May 2019.

[95] P. Association, *ISO 32000-2*, 2017.

[96] T. Lee, "What Gets Redacted in Pacer?" https://freedom-to-tinker.com/2011/06/16/what-gets-redacted-pacer/, Freedom to Tinker, 2011.

[97] E. Sandhaus, "The New York Times Annotated Corpus," https://catalog.ldc.upenn.edu/LDC2008T19, Linguistic Data Consortium, 2008.

[98] "North Carolina Voter Data," https://www.ncsbe.gov/results-data/voter-registration-data, North Carolina State Board of Elections, 2022.

[99] "Ohio Voter Data Download," https://www6.ohiosos.gov/ords/f?p=VOTERFTP:STWD:::#stwdVtrFiles, Ohio Secretary of State, 2022.

[100] "Washington Voter Registration Database Extract," https://www.sos.wa.gov/elections/vrdb/extract-requests.aspx, Washington State Office of the Secretary of State, 2022.

[101] "Redacting with Confidence: How to Safely Publish Sanitized Reports Converted From Word to PDF," 2005.

[102] O. N. Romanyuk, S. V. Pavlov, O. V. Melnyk, S. O. Romanyuk, A. Smolarz, and M. Bazarova, "Method of Anti-Aliasing with the Use of the New Pixel Model," in *Optical Fibers and Their Applications 2015*, vol. 9816. SPIE, 2015, pp. 274–278.

[103] Y. Zhou and D. Feng, "Side-Channel Attacks: Ten Years After its Publication and the Impacts on Cryptographic Module Security Testing," *Cryptology ePrint Archive*, 2005.

[104] D. Lopresti and A. L. Spitz, "Quantifying Information Leakage in Document Redaction," in *Proceedings of the 1st ACM workshop on Hardcopy document processing*, 2004, pp. 63–69.

[105] C. Whelan and D. Naccache, "US Intelligence Exposed as Student Decodes Iraq Memo," *Nature*, vol. 429, no. 6988, pp. 116–116, 2004.

[106] C. Fan, B. Qi, S. Mitra, and M. Viswanathan, "DryVR: Data-Driven Verification and Compositional Reasoning for Automotive Systems," in *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*. Springer, 2017, pp. 441–461.

[107] F. Wang and Y. Shoshitaishvili, "Angr-The Next Generation of Binary Analysis," in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 8–9.

[108] S. O. Madgwick, A. J. Harrison, and R. Vaidyanathan, "Estimation of IMU and MARG Orientation Using a Gradient Descent Algorithm," in *2011 IEEE international conference on rehabilitation robotics*. IEEE, 2011, pp. 1–7.

[109] "eYSIP-2017 Quadcopter firmware," 2017, https://github.com/heethesh/eYSIP-2017_Control_and_Algorithms_development_for_Quadcopter.

[110] A. Keliris and M. Maniatakos, "ICSREF: A Framework for Automated Reverse Engineering of Industrial Control Systems Binaries," in *Network and Distributed System Security Symposium (NDSS)*, 2019.

[111] D. Currie, X. Feng, M. Fujita, A. J. Hu, M. Kwan, and S. Rajan, "Embedded Software Verification Using Symbolic Execution and Uninterpreted Functions," *International Journal of Parallel Programming*, vol. 34, no. 1, pp. 61–91, 2006.

[112] O. Sery, G. Fedyukovich, and N. Sharygina, "FunFrog: Bounded Model Checking with Interpolation-Based Function Summarization," in *Automated Technology for Verification and Analysis: 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings 10*. Springer, 2012, pp. 203–207.

[113] P. R. Gluck and G. J. Holzmann, "Using SPIN Model Checking for Flight Software Verification," in *Proceedings, IEEE Aerospace Conference*, vol. 1. IEEE, 2002, pp. 1–1.

[114] D. Babić and A. J. Hu, "Structural Abstraction of Software Verification Conditions," in *International Conference on Computer Aided Verification*. Springer, 2007, pp. 366–378.

[115] M. Aizatulin, A. D. Gordon, and J. Jürjens, "Extracting and Verifying Cryptographic Models from C Protocol Code by Symbolic Execution," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 331–340.

[116] J. Park, M. Pajic, O. Sokolsky, and I. Lee, "Automatic verification of finite precision implementations of linear controllers," in *Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I 23*. Springer, 2017, pp. 153–169.

[117] D. Jackson and A. Waingold, "Lightweight Extraction of Object Models from Bytecode," in *Proceedings of the 21st International Conference on Software Engineering*, ser. ICSE '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 194–202.

[118] G. Ramalingam, R. Komondoor, J. Field, and S. Sinha, "Semantics-Based Reverse Engineering of Object-Oriented Data Models," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 192–201.

[119] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Păsăreanu, H. Zheng, and W. Visser, "Tool-Supported Program Abstraction for Finite-State Verification," in *Proceedings of the 23rd International Conference on Software Engineering*, ser. ICSE '01. USA: IEEE Computer Society, 2001, p. 177–187.

[120] R. Ji, R. Hähnle, and R. Bubel, "Program Transformation Based on Symbolic Execution and Deduction," in *Proceedings of the 11th International Conference on Software Engineering and Formal Methods - Volume 8137*, ser. SEFM 2013. Berlin, Heidelberg: Springer-Verlag, 2013, p. 289–304.

[121] M. Trtik and J. Strejcek, "Symbolic Memory With Pointers," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2014, pp. 380–395.

[122] D. Engler and D. Dunbar, "Under-Constrained Execution: Making Automatic Code Destruction Easy and Scalable," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 1–4.

[123] E. Coppa, D. C. D'Elia, and C. Demetrescu, "Rethinking Pointer Reasoning in Symbolic Execution," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 613–618.

[124] M. Hind, "Pointer Analysis: Haven't We Solved this Problem Yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2001, pp. 54–61.

[125] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song, "Learning Loop Invariants for Program Verification," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[126] N. L. Ricker, "Decentralized Control of the Tennessee Eastman Challenge Process," *Journal of process control*, vol. 6, no. 4, pp. 205–221, 1996.

[127] A. Keliris, H. Salehghaffari, B. Cairl, P. Krishnamurthy, M. Maniatakos, and F. Khorrami, "Machine Learning-based Defense Against Process-aware Attacks on Industrial Control Systems," in *2016 IEEE International Test Conference (ITC)*. IEEE, 2016, pp. 1–10.

[128] E. Johnson, M. Bland, Y. Zhu, J. Mason, S. Checkoway, S. Savage, and K. Levchenko, "Jetset: Targeted Firmware Rehosting for Embedded Systems," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 321–338.

[129] B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1237–1254.

[130] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1201–1218.

[131] M. Baezner and P. Robin, "Stuxnet," ETH Zurich, Tech. Rep., 2017.