Proposal for Ph.D. Dissertation. Maxwell Bland, February 2023

Overview:

This dissertation will synthesize and systematize the knowledge of discoveries made during three diverse projects in program analysis. It will unite these projects' findings into a cohesive narrative and present hereto unaddressed technical details, algorithms, and methodological steps adopted but not discussed within the three academic articles due to space constraints. These three projects have resulted in both accepted peer-reviewed conference publications at USENIX Security and the Privacy Enhancing Technologies Symposium (PETS), and an in-submission publication to the Conference on Computer Aided Verification (CAV).

The present author was responsible for a significant part of the implementation of the Jetset firmware rehosting system [1]. The present author's contributions included techniques and code central to the work, such as dynamic callchain resolution and the implementation of a rudimentary symbolic execution system based upon taint tracking with the QEMU embedded device emulator [2]. The author's contributions also included the development of one of the first architecture independent full-system fuzzers for embedded systems requiring no modification to the target firmware. The work led to the discovery of over 200 faulting system call codepaths within the Communication Management Unit 900 (CMU-900) used by Boeing 737 aircraft. The author used these codepaths and a sophisticated code upload attack to write and confirm a privilege escalation exploit on both an emulated and physical version of the CMU-900.

Second, the author was responsible for the implementation, evaluation, and discovery of novel deredaction attacks on PDF documents through the use of models of "glyph shifting scheme" algorithms used by popular document editors (Microsoft Word and Adobe Acrobat). The attacks in question leveraged novel information leaks that rendered the redaction of names from several documents of historical relevance ineffective. The models of algorithms in question were recovered using a key insight: the combination of single core, virtualized execution with recent innovations in time-travel debugging [3]. The work resulted in the discovery of hundreds of newly vulnerable redactions, the discovery of an entire new class of text redaction vulnerabilities, and significant efforts on behalf of affected parties to change present text redaction practice.

Third, the author was the primary contributor to a novel lifting approach extending the abilities of existing binary firmware symbolic executors, such as angr [4], to recover continuous control equations from symbol-stripped firmware. By soundly and automatically translating arbitrary program slices into symbols for uninterpreted functions, the InteGreat system is able to decompile real-world, symbol-stripped ARM assembly into Matlab Simulink functions verifiable in some continuous model of the environment. The system was used to find differences (a bug) in the implementation of a published quad-copter orientation estimation algorithm, reproduce and model the effects of a code upload attack on the reactor pressure of a chemical plant, and identify key limitations to the emulations synthesized by firmware rehosting systems.

Intellectual Merit:

The proposed dissertation will provide commentary and a cohesive narrative to the three works in question. It will also include valuable technical details and insight into the implementation of the three systems and the deeper problems of program analysis encountered in the study of embedded systems.

Broader Impacts:

The works have led to responsible vulnerability disclosure to UTC Aerospace, the stakeholder of the CMU-900, and the disclosure of significant numbers of broken redactions to government agencies, and patentable code artifacts for use by the Department of Energy. The proposed dissertation will include further work to cement the impact of discovered redaction vulnerabilities and the continuous equation lifting artifact.

1 Introduction

The proposed dissertation will systematize and address the strengths and weaknesses of emulation, dynamic analysis, modeling, and decompilation in understanding the semantics of symbol-stripped binary code. More generally, it will provide empirically backed perspectives on the challenge of writing algorithms to analyze other algorithms under information loss conditions, drawing on four years of research in program analysis.

The first discussion centers on the domain of *firmware rehosting* and Jetset, accepted to USENIX 2021, which allows codes in a specific physical domain, e.g. an ARM SoC with hardware sensors for controlling a quad-copter, to be executed and analyzed in another domain, e.g. a laptop computer. Critically, this binary code lacks string identifiers that would aid understanding and this firmware's analysis often encounters hard problems, such as the necessity loop invariant inference. THe primary goal of rehosting includes dynamic analysis of the a firmware image using existing offensive techniques, such as fuzzers, which cannot be carried out on the physical machine due to the possibility of damaging hardware or due to limitations on the capability of executing code. *The Jetset work led to the discovery and reproduction of a privilege escalation attack on the CMU-900's VRTX operating system.*

In discussing rehosting, the proposed dissertation will provide a greater level of technical depth and explanation on the Jetset system's approach to symbolic execution, namely, the use of symbolic execution in the inference of necessary hardware semantics for bootstrapping an emulation of a firmware image. It will discuss the limitations of existing abstract interpretation approaches and technologies, including those used in the paper, and identify concepts essential the abstract interpretation of binary code. The proposed dissertation will also discuss the issue of fuzzing rehosted systems and embedded systems in general, a problem that continues to see active research [5]. More essentially, however, Jetset introduced a specific problem: the precise modeling microarchitectural semantics and the algorithms embedded in binary code.

As a result, the present author proceeded to begin work on the precise modeling of symbol stripped binary programs, resulting in *Story Beyond the Eye*, accepted to PETS 2023. This work targeted the domain of PDF text redaction security in particular. When compared to rehosting, the exact modeling the algorithms embedded in binary code was discovered to be more powerful for certain applications. *It was found that by exactly modeling the glyph shifting schemes of PDF document producers, novel information leaks could be "fingerprinted" and used to exactly de-redact the names of individuals excised from the PDF.*

The proposed dissertation will explore the extraction of algorithms for PDF document production from standard executables. It will also detail the differences between the extraction of process traces from an embedded system like the CMU-900 and from standard executables. Notably, it will discuss technical details involved in the dynamic analysis of embedded systems, including the necessity of trampoline code and the complexities of relocation created by the problem of binary code rewriting [6]. The utility and specifics of time-travel debugging, a form of data-flow analysis based upon stepping backwards and forwards in a program's execution, will also be made more explicit than in published results. Reverse engineering was not the focus of *Story Beyond the Eye*, despite this being the primary technical overhead, due to the novelty of the discovered attack. Limitations, such as the fact that manual effort is required to perform the extraction of glyph shifting algorithms and only a single execution trace is recorded, will be made explicit. This discussion provides future analysts and researchers with a blueprint for extracting glyph shifting schemes and exact reproductions of other algorithms from binary code, embedded or otherwise.

The results of both of these prior works in rehosting and algorithm extraction suggested a more general solution to several of the difficulties of binary program analysis. In particular, there was an explicit need for a specification language and system allowing analysts to abstract arbitrary program slices into the domain of theorem provers like Z3 [7].

The dissertation therefore next addresses the InteGreat system, in submission to CAV 2023, which allows researchers to lift precise models of algorithms from embedded binary code. The system automates several of the difficult, manual analysis steps encountered when attempting to extract algorithms from closed source binaries through an object-oriented framework for program slice abstraction (function summarization [8]). *In particular, this lifting, when applied to a PLC, was useful in exactly reproducing and analyzing a code-upload attack precisely destabilizing the reactor pressure of a Eastman-Kodak chemical plant.*

Summarizing, the key contribution of the proposed dissertation is therefore a set of empirically justified statements on potential solutions to practical problems encountered when performing binary program analysis, given the empirical perspective provided by three academic works. While not noted in this introduction, the findings also provide a well-supported argument for continued work on lifting systems and may help the uninitiated understand how computers construct meaning from symbols. The work also provides significant insights into the concept of *information loss*, both at an abstract level (e.g. when text is redacted) and at a concrete level (e.g. when a variable's name is obsfucated during compilation).

The work that composes the proposed dissertation has had immediate, broad impacts. Jetset's core finding, an exploit for the CMU-900's operating system, resulted in direct communications with avionics manufacturers on the security of their systems. The work on deredaction resulted in the discovery of hundreds of broken redactions, notifications of several affected parties, including the US Courts, and actions on behalf of several of these parties to prevent future information leaks. Moreover, the InteGreat work has already begun to affect the direction of firmware rehosting research inside the Department of Energy.

The contributions of the proposed dissertation will be:

- A detailed technical explication of the techniques used to achieve significant results in three academic works, including the use of symbolic execution in firmware rehosting, full-system embedded firmware fuzzing, methodolgy for the extraction of glyph shifting algorithms for the purpose of breaking PDF text redactions, and the implementation of a framework for lifting continuous control equations from symbol-stripped binaries.
- Empirical results attesting to the (in)effectiveness of certain solutions to the problems program analysis. These problems include path explosion during symbolic execution, the interpretation and modeling of microarchitectural semantics, and information loss during the execution, compilation, and decompilation of programs.
- A synthesis of the concepts from otherwise disconnected, complex research works into a complete whole, providing a detailed narrative of the contemporary research landscape as it relates to systems for symbol-stripped binary code analysis and abstract interpretation of firmware images.

2 Background

This section addresses the necessary background to understand the methodologies present in Section 3. We begin by disucssing related work in firmware rehosting and the analysis of embedded systems. Following from this, there is a brief discussion of deredaction in the context of reverse engineering—primarily, the recovery of algorithms involved in the specification of PDF documents. Finally, we end with prior work in the verification and abstract interpretation of binary code semantics.

2.1 Firmware Rehosting and Analysis

The process of understanding or attacking a system like the Communication Management Unit of a Boeing 737 typically begins with the extraction of the firmware, or binary code, from hardware (flash units) storing this data, either via desoldering or via the use of wired connections [9]. Once this code is extracted, it is typically brought into a decompiler, such as Ghidra [10], which lifts the binary code back into a pseudo-C representation from an intermediate representation. This intermediate representation is typically a language with semantics that allow the decompiler to target diverse microarchitectures and allow algorithms operating on the intermediate representation to more easily perform decompilation [11]. The recovered pseudo-C representation may then be reverse engineered and experimented with by researchers to better understand how the firmware functions.

However, the process of decompilation itself involves several concepts of importance to the proposed dissertations, and continues to be an active area of research [12]. Of these, the proposed work focuses on *abstract interpretation*, introduced by Cousot and Cousot around 1977 [13], a model for the static analysis of programs by the construction or approximation of fixpoints, or lifting (microarchitectural) semantics from concrete to abstract domains.

The abstract interpretation of binary programs has several equivalents, and has deeper ties into the notion of data-flow recovery and taint-tracking [14]. Abstract interpretation, when applied to binary code, is often referred to as symbolic execution, for which there are several notable systems, with more popular recent examples including KLEE and angr [15, 4] These systems often work by recording symbolic variants of concrete operations. For example, the statment $(x > 5)?\, y+1 : x+1$ would be concretely evaluated to 4 if $x = 3$, however, under 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. A symbolic executor and other routines for static analysis are present in the Ghidra decompiler, in order to provide an accurate and reconfigurable reverse engineering and decompilation environment.

Emulation and Dynamic Analysis. One reasonable next step, beyond the decompilation of the program, is to attempt to *run* the program and analyze this execution, a subdomain referred to as dynamic analysis, introduced by Ball in 1999 [16]. For contexts in which the binary code under evaluation may be run on readily available hardware with the capability for instrumentation, it can be debugged, the execution may be traced and evaluated, or *fuzzed*, inputs can be given to the system in an intelligent or random manner in order to fully evaluate the behavior of said system [17].

However, it is often *not* the case that a binary program can be executed in an environment that allows for instrumentation and dynamic analysis. Emulators, such as QEMU [2], attempt to correct for this by reproducing the context the original binary code was expecting to be emulated within, by simulating hardware interactions and by reinterpreting instructions into an microarchitecture-independent intermediate representation that may be executed on a virtual machine.

Rehosting. The implementation of emulators is often an arduous process, involving careful study and a precise understand of the target system and hardware context. The domain of *firmware rehosting* attempts to automate this process, though various approaches. Both FIE [18] and Jetset [1] attempted to use a symbolic execution to bootstrap emulators for embedded firmware: they generate symbolic constraints for algorithms the firmware uses to interact with hardware, identify execution paths leading to a desired point in the firmware's execution, and then solve the constraint sets for that path in order to recover the precise memory reads necessary to trigger that path. Firmadyne [19] and Costin et al. [20, 21], incontrast, attempted to just match the hardware interaction constraints of specific software, i.e. the Linux kernel and networking stacks. Pretender [22] attempted to rehost firmware by recording the interactions between the physical hardware and the firmware. HALucinator [23] 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 replace it with handlers that properly simulate the hardware interaction. P^2IM [24] performed fuzzing to identify the correct hardware read values necessary to trigger a particular program path.

The ultimate result of rehosting is, ideally, a system capable of executing the firmware and reproducing key system behaviors, e.g. a system that can be accurately fuzzed. Such a digital twin is also useful in several contexts, including the generation of digital twin systems that replicate the functioning of a complex cyberphysical system and its environment in software. However, the domain still faces several challenges, including fidelity, firmware acquisition, static analysis for bootstrapping rehosting systems, parallelized emulation, and even after successful identification, vulnerability identification and integration of the emulation into other systems [25].

The proposed disseration will shed further light on the problem of rehosting introduced by these prior works, hinted at in the methodologies of Sec. 3, and serve to further connect the problems of rehosting to the problems of abstract interpretation and meaning-making of binary codes in the presence of lost information (e.g. hardware components, symbols).

2.2 Deredaction and Information Leaks

Rehosting is related to deredaction in that both problems relate to the recovery of lost information. In the former case, the function of a system in an original, physical context, and in the latter, a removed text. To identify these parallels and better understand information loss, the proposed dissertation next addresses the problem of *deredaction*, with particular focus on PDF text.

When text is removed from a document in the classical sense, using a black marker or white-out, the width of the text is still observable given the surrounding words [26]. Considering this alone as a *perfect* process on a monospaced font, the words “cat” and “dog” become indistiguishable. However, the information loss is almost never perfect: for example, in Times New Roman, the words “cat” and “dog”, if redacted, are distinguishable by their widths. Information leaks present in redacted PDF documents were identified by Lopresti and Spitz [27], who developed a system for breaking redactions where the precise TTF width was known.

However, the Lopresti and Spitz work ended up failing to account for several important aspect of the problem: first that a rasterization workflow may change a PDF document’s glyph positioning and physical printing may not be a pixel-perfect reproduction of the digital document, and second, that TTF glyph widths do not necessarily equate with PDF document or raster glyph widths. Moreover, they missed an additional, severe facet of the problem: that in PDF documents, width alone was not being leaked: there are also sub-pixel sized shifts applied to non-redacted glyphs that can be *dependent* on redacted glyphs’ data. This was the subject of *Story Beyond the Eye*, a work narrativized by the proposed dissertation [28].

It is useful to consider, then, cases where an information leak is not captured and cases where it is misinterpreted. There is a wealth of literature on the improper removal of information from digital files that follows this pattern, and *Story Beyond the Eye* also, understandably, only partially solves the problem. Forrester and Irwin [29] discuss nonexcising redactions and unscrubbed metadata such as the Producer field of PDF documents but do not mention glyph positioning based deredaction. Hill et al., used hidden Markov models to recover text obscured either by mosaic pixelization or a related tactic, e.g. Gaussian Blur [30], but fail to deredact text obscured by a single box. While Müller et al. [31] discuss hidden information present in PDF documents, specifically PDF document revision information and author name metadata, but do not explicitly tackle redaction. Other file formats may also be deredacted: Murdoch and Dornseif [32] discuss how cropped JPEGs can preserve uncropped image information, but these works do not dicuss text redaction in particular.

Irrespective of the studied medium (PDF redaction, JPEG redaction) information left or destroyed is always a result of a communication system [33], and in most cases this data flow is determined by a software system—by binary code interpreting binary code. Thus the proposed dissertation will highlight a portion of *Story Beyond the Eye* not discussed in the publication, the extraction of exact models of PDF text positioning algorithms from closed-source software.

2.3 Function Summarization and Code Lifting

As a result of similar problems to those encountered in *Story Beyond the Eye*, there is a wealth of research on the subject of *function summarization* and lifting of firmware binaries, particularly within the software verification and reverse engineering communities. The immediate form of this problem is software clone identification, which attempts to find identical program slices across binaries [34], while the theoretical landscape stretches to the generation of a precise abstract interpreter from a system of logic [35]. *Function summarization* is defined as follows [36]:

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.

The necessity of loop unrolling here is somewhat strict and can be replaced by inferred invariants [37]. *Binary lifting* raises machine instructions to higher-level intermediate representations (IR) such as LLVM [38, 39, 40]. The value provided by summarization and lifting is the precise association and identification of useful information within a binary program.

The final work the proposed dissertation integrates, InteGreat [41], reinterprets bitvector-domain symbolic execution into the theory of uninterpreted functions to perform modular, nestable function summarization and decompilation. It provides a specification language that allows users to abstract arbitrary program slices in symbol-stripped binary code with symbols and statements in systems of logic.

A long line of work has utilized symbolic execution to perform model extraction and subsequently verification on the extracted models. SPIN [42], defined the term “model extraction” and applied model-checking on aero-space flight software. Babić and Hu [43] used natural abstraction boundary identification and symbolic execution to optimize the performance of verification. Hernandez et al. [44] and [45] 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 [46]. Jackson and Woodward [47] extracted object-oriented (OO) models from Java bytecode, [48] extracted OO data models from weakly-typed source code. Bandera [49] 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 these techniques could improve InteGreat, prior work does not address the possibility of a generic language for the specification of these abstraction extraction boundaries, and does not solve the specific problems involved in stiching together uninterpreted functions as abstractions.

Ji et al. [50] perform backward application of extended sequent calculus rules on symbolic expression trees. InteGreat’s approach, abstraction resolution, is similar to a sequent calculus approach. However, this work symbolically executes by sequent calculus rules and does not attempt summarization. Instead, the goal of the work is bisimulation and optimization of the analyzed algorithms.

The closest work to InteGreat is Currie et al. [51]. Currie et al. *only* consider the problem of equivalent programs across compiler optimizations, and do not use uninterpreted functions to target *decompilation*, only to check equality. This is a stictly easier problem than the one we solve, because we address the necessity of retaining viable semantics before and after skipping a program slice (by identifying side-effects of the removed slice automatically).

The proposed dissertation will offer greater technical explanations of the core algorithms used by InteGreat and analyze the relationship between this work and that of Jetset and *Story Beyond the Eye*. It will consider how the sound substitution of a program slice's semantics via abstract interpretation, the core technical feat of InteGreat, relates to the process of information loss in transformations of digital data. In particular, by presenting InteGreat's use of deductive specifications in the problem of abstraction, the paper will present one perspective on the nature of information loss in digital systems.

2.4 Remaining Problems

Because this dissertation will draw conclusions from published work and work in submission, it will only include novel experiments insofar as is necessary to justify claims made in the content of the dissertation, where existing empirical evidence will not suffice. Unresolved domains of the proposed dissertation will include cases where the binary code under evaluation is not related to either an embedded system, a digital document, or a control equation that may be more readily abstracted than arbitrary computation. The nature of these experiments and claims, due to realistic limitations on the grounds with which they may be justified, will be intimately related to the theoretical and real capacities of Jetset, *Story Beyond the Eye*, and InteGreat. Where appropriate the proposed dissertation will detail future work *specific to the subdomain* of each work.

To provide a listing of just a few such specific future works, this includes, for example, limitations on the capabilities of existing symbolic executors to control the state space explosion encountered when attempting to interpret loops. It also includes difficulties in resolving pointer aliasing in symbol-stripped firmware, and lack of support within current symbolic executors for detailed support of extra-assembly semantics, such as task switching and interrupts. For redaction, it includes issues relating to the resolution of precise glyph shifting schemes used within printed or rasterized documents, the implementation or recovery of additional glyph shifting schemes, and the generic tracing of specific PDF production workflows. With respect to the InteGreat work, this includes mathematical abstractions for guaranteeing the correctness of transformations between one semantic domain and an arbitrary, more abstract domain, and subsidiary issues, such as the perfect identification of a given target semantics within a symbol-stripped binary program.

3 Research Methodology

The following section introduces the methodologies and technical contributions of the Jetset, *Story Beyond the Eye*, and InteGreat works. In each subsection, an overview of the work is provided and then followed by an explanation of the specific technical contributions the dissertation will expand upon and discuss.

3.1 Jetset

The Jetset system uses symbolic execution to infer what behavior firmware expects from a target hardware device. It recovers traces of the expected I/O behavior, and then generates device models for hardware peripherals in C, allowing the an analyst to boot the firmware in an emulator. Jetset was applied to 13 firmware images, the most complex of which were a Feeder Protection Relay, the Communication Management Unit of a Boeing 737, and a Raspberry Pi. The emulated firmware image also facilitates fuzz-testing, and a custom, architecture-independent fuzzer for QEMU, requiring no firmware modification, was used to discover a previously unknown privilege escalation vulnerability.

3.1.1 Techniques

The core of Jetset is composed into three parts: *specification*, *peripheral inference and synthesis* and *search strategy*. The *specification* fed to Jetset consists of four parts, which must be attained via alternative means:

- The **binary code** of the target, which must be read out of a physical flash or extracted from the target by some other means.
- The **memory layout** of the target, which specifies how physical devices are mapped into the memory of the device, and is used to determine which reads and writes are key to peripheral inference.
- The **entry point** of the target, e.g. where execution begins. In this sense, Jetset attempts a *brute force* inference of the target, by beginning inference from the absolute start of execution. This stands in contrast to InteGreat, discussed later.
- A **goal address** to reach within the target. Jetset explores *a single* execution path to a specific firmware state, and employs a *single* search strategy to get to this target. This again will stand in contrast to InteGreat.

This specification is then used to begin the process of *peripheral inference and synthesis*, the former of which consists of symbolically executing the firmware to infer what values should be returned by reads from device registers. The symbolic execution uses a taint-tracking system build on top of the QEMU TCG intermediate representation, paired with the angr constraint solver, to determine which read values are necessary to achieve a particular execution trace. This union between taint tracking and symbolic execution was noted in [14]. Jetset and the popular SymQEMU [52], avoid supporting hundreds of essential TCG helper methods which capture important microarchitectural semantics, such as task switching, in order to make useful theoretical discoveries. Four attempts to elegantly address this problem culminated in the later discovery and implementation of the InteGreat system.

The *synthesis stage* for peripherals recovers the specific model by making a call into angr's wrapper around the Z3 solver to find a concrete value that satisfies the constraints necessary to execute the desired program path. For the purposes of finding a goal location in the firmware, Jetset naively applies just a single *search strategy* to locate a target execution point, a combination of Tabu search, a variation of depth-first search guided by a distance function with a list of "avoid" states and a context-sensitive distance function generated by resolving the callgraph of dynamically resolved function calls during symbolic execution. The latter is achieved by "stitching" the discovered call target address during the symbolic execution into the recovered control flow graph so far. An overview of this process is given in Fig. 1.

By adding the context of the child-call CFG as calls are resolved at runtime, Jetset takes into account the fact that the distance between two instructions in a program can depend on the calling context, i.e. the call-stack of the two instructions. Because programs may modify their own callstack to perform returns to different locations, distance from the goal location is computed lazily from the actual stack during execution. When Jetset is not able to resolve a path to the target due to indirect or runtime-calculated function pointers, it alternates between resolving branches as *true* or *false*, allowing it to escape from exploring the same loop resolution condition infinitely.

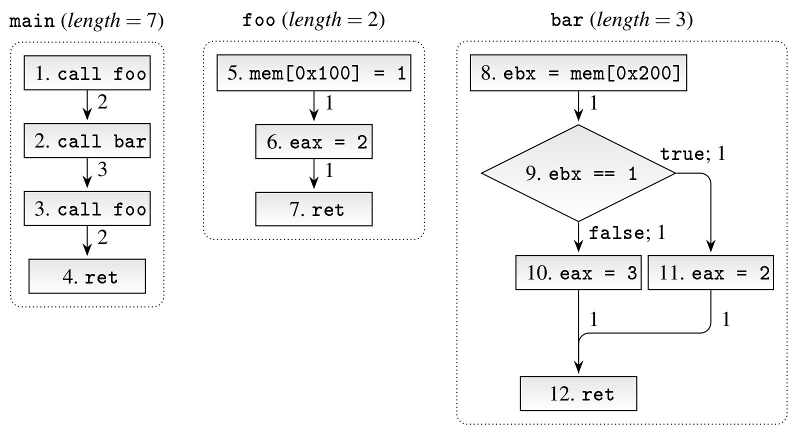


Figure 1: Context-sensitive distance from statement 5 (in first `foo` call) to statement 7 (of second `foo` call).

3.1.2 Evaluation

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 Collins Aerospace CMU-900, an electronic system 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 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.

While it was not addressed in the original publication, the testing of these systems also involved significant amounts of fuzz-testing using a custom built fuzzer that integrated AFL and QEMU, similar to FirmAFL [53], but without requiring the modification of the target program. It achieved this feat by performing a process-tree snapshot of the entire QEMU process while preserving speed by mounting QEMU's files into a RAM filesystem. In the proposed dissertation, more detail on this fuzzer's implementation and the technical novelties would be provided—valuable due to the fact that the fuzzer has seen continued use after the paper's publication by a number of unaffiliated researchers.

The two targets of fuzzing 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.

	Raspberry Pi 2	**CMU-900**	**SEL-751**
CPU/SoC	Broadcom BCM2836 (ARM)	AMD Am486 (i386)	NXP MCF54455 (ColdFire)
OS/SW	Linux 4.19.y	VRTX-32	G5.1.5.0
Peripheral inference			
Wall-clock time	6m43s	5m20s	2h34m51s
Blocks in code base	238,792	55,016	141,750
Total blocks executed	81,194,393	53,143,508	3,351,484,857
Blocks executed on path	81,194,393	27,517,932	3,351,484,857
Unique blocks executed	43,157	776	11,364
Unique blocks executed on path	43,157	731	11,364
MMIO writes (ignored) on path	84,060	1,308	32,480
MMIO reads (symbolic) on path	83,857	242	704
MMIO write addresses on path	40	13	68
MMIO read addresses on path	37	5	26
Devices accessed	6	5	5
Peripheral synthesis			
Wall-clock time	3.16s	0.018s	5.61s
Total Symbolic Variables	1,384	242	704
Total Constraints	5,226	756	11,142
Constraints per variable	3.78	3.12	15.83
Average trace length	37.4	48.4	27.08
Median trace length	1	5	2
Maximum trace length	1076	215	343
Emulator execution to goal			
Wall-clock time	8s	289ms	1m1s
Total blocks executed	81,454,594	27,519,080	3,351,502,947
Unique blocks executed	43,255	731	11,364
MMIO writes (ignored)	83,915	1,882	32,480
MMIO reads	83,857	242	704
MMIO write addresses	43	13	68
MMIO read addresses	27	5	26
Devices accessed	6	5	5

Table 1: Evaluation targets and summary statistics.

To validate this exploit on the physical device, for which the firmware version was different, the author had to build a ground truth QEMU emulation for the second firmware, and then write dataloading code to upload an handcrafted, shimmed version of the VRTX operating system which could inject the malicious system call at a realistic point in the physical device's execution. This process was successful, and due to the complexity of relocating the shim code on the memory-constrained physical device while maintaining a true-to-reality execution of the system, required hundreds of lines of hand-written x86 assembly.

The discovery and crafting of this exploit would be further elucidated and detailed within the proposed thesis, and is valuable as it later led to the discovery of three remote denial-of-service vulnerabilities on the CMU-900 (unpublished), that cause the machine to crash due to maliciously crafted aircraft communication and reporting system (ACARS) messages.[1]

[1]Because ACARS messages also have a "broadcast" mode, these messages couldbe used to crash a large number of airplanes' CMU-900's simultaneously.

3.1.3 Discussion

The Jetset work was a novel contribution to a young domain of firmware rehosting. Thus, the techniques and methodologies adopted during symbolic execution and fuzzing had limitations. For one, the resulting emulation was not perfect. However, a perfect emulation was not necessary for useful results. While Jetset only explored and inferred constraints for a single execution path, the results of fuzzing these emulations led to a real vulnerability that was disclosed to the embedded system's vendor.

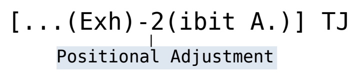
The current technical landscape of of symbolic execution and fuzzing on embedded systems is severely limited by support for microarchitectural abstractions, as, to the author's knowledge, all current embedded system fuzzers require modifications to the firmware under evaluation, and all symbolic executors fail to support all complex processor features, e.g. x86 task switching. The proposed dissertation will address solutions to this limitation, both in the presentation of InteGreat and in detailing the construction of yAFL, the fuzzer Jetset used to discover the CMU-900 vulnerability.

3.2 Story Beyond the Eye

In *Story Beyond the Eye* we found that many current redactions of PDF text are insecure due to non-redacted character positioning information, determined by the *glyph shifting* algorithms PDF producers use. Subpixel-sized horizontal shifts in redacted and non-redacted characters can be recovered and used to effectively *deredact* first and last names—these shifts add additional information to the width of redacted information and can be dependent upon redacted information but not themselves redacted. These findings affected redactions where the text underneath the black box is removed from the PDF.

The system uses models of glyph shifting algorithms to correctly fingerprint the information left by redacted text, and was able to "break" hundreds of real-world PDF redactions, including in documents of historical relevance, OIG investigation reports, and FOIA responses. The work also included an extensive notification of affected parties, demonstrating the broad impact of the work.

3.2.1 Techniques



```
[...(Exh)-2(ibit A.)] TJ
```

Figure 2: The TJ text showing operator, which specifies the glyphs to render and, by reference to a font object (not shown), their widths, along with any associated positional adjustments, given in text space units.

The significant discovery of *Story Beyond the Eye* was the existence and utilization of a novel redacted text information leak. PDF documents can render text in numerous ways, including by use of a text showing operator, one of which (TJ) is depicted in Figure 2. 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.

Glyph advance widths and glyph shifts create a security concern, and *Story Beyond the Eye* found that most PDF redaction tools replace text selected for redaction with a single large shift of the same width as the redacted text showing operator, creating the two significant security risks:

```
 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 (((disp > 0.003) || (disp < -0.003)) && i != vs->size() - 1) {
 8       int adj = disp * 1000 + 0.5;
 9       vs->setShift(j, adj);
10       ttf = msWord = 0;
11     } else {
12       vs->setShift(j, 0);
13     }
14 }
```

Figure 3: Snippet of reverse engineered code representing how Microsoft Word leaks redacted character information into non-redacted characters in a PDF document.

- The precise width of the redaction can be used to eliminate potential redacted texts, and is made more distinct than advance widths alone by glyph shifts.
- Any non-redacted glyph shifts conditioned on redacted glyphs can be used to eliminate potential redacted texts.

The work also addressed concerns related to nonexcising redactions. These redactions are cases where the text underneath the redaction can be selected and copied to the system clipboard from the PDF document.

Glyph shifts may then be classified as *independent* or *dependent*, where the former implies they are not determined by any particular character in the document, and the latter implies that they are—this is precisely dangerous if the character they are conditioned on is redacted. This itself is not dangerous if the amount of information leaked on redacted text is small or the redaction tools themselves remove this information. Thus, the methodology of the paper was split into extracting and evaluating glyph shifting schemes, e.g. from Microsoft Word's "Save-as-PDF" feature, and evaluating the information removed by redaction tools.

The latter corresponded to an evaluation of what types of information the redaction tools leaked, and after reverse engineering the schemes produced by 11 redaction tools, including Adobe Acrobat, the paper finds *none* but those that rasterize the document entirely sufficiently mitigate these leaks. Two of the tools were completely broken, and did not remove the text at all (created nonexcising redactions): the present author notified them of this problem and as a result of this both the tools have published patches.

The extraction and evaluation of the glyph shifting schemes involved precise tracing of PDF producer software and reverse engineering to extract an exact model of their positioning algorithms for text. Microsoft Word, in particular, provided glyph shift values that were *highly* dependent on redacted glyphs, due to a floating-point error accumulation algorithm that compares the "real" PDF position of a glyph with a set of artificial positions determined by the full text of a given line. A portion of this behavior, one of the error accumulators, is presented in Fig. 3. Note the internal widths used on line 3 of the figure are determined by a loop with no overflow reset and the redacted information held by the accumulator is not zero after a single shift is written: this detail is given in the publication's appendix.

The extraction of the prior algoithm and reverse engineering of PDF producer structures in a rich area and was not given sufficient space due to constraints on the original publication. The proposed dissertation would provide both an explanation of how these algorithms were extracted as well as some of the chal-

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

Table 2: Dictionaries containing candidate texts used for evaluating deredaction. Stop words are excluded from the statistics.

lenges and solutions developed when addressing the problems of discovering and breaking redactions, and classifying glyph shifting schemes.

3.2.2 Evaluation

Synthetic Evaluation. The evaluation of the vulnerability these shifting schemes create for redacted text was based on the simulation of redactions on text from the New York Times annotated corpus [54] using various *dictionaries*. This represents how the amount of information leaked 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. Notably, these dictionaries included:

- *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 [55], Ohio [56], and Washington [57] were chosen based upon the availability of publicly accessible data.

The amount of information leaked in the simulated redaction was measured both upon a uniform ($H_u(X)$) and empirical ($H_e(X)$) frequency distribution of each dictionary entry, i.e. one wherein the adversary knows which dictionary entries are more likely to occur and one wherein they treat each dictionary entry as equally possible. It is therefore important to understand the size of these dictionaries and the number of bits of information they contain *in total* with respect to both uniform and empirical distributions, so

		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%

Table 3: 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. Refer to Table 2 for the total number of bits of information present in the candidate dictionary. "Probability correct guess" refers to the likelihood of randomly selecting the redacted word given the (typically small) set of matching candidate texts.

that the following synthetic evaluation results may be placed in context. Scale measurements are given in Table 2.

The results of this synthetic evaluation are given in Table 3. Note that four different fonts and three glyph shifting schemes were evaluated: Courier presents a strawman result for a purely monospaced font with no shifts. The *Un* column presents the efficacy of deredaction provided an attack utilizing the width of the PDF text alone, if no adjustments occur. This is a specific case of an *independent* shifting scheme. The *W07* and *W19* columns present the Microsoft Word shifting schemes in the 2007 and 2019 desktop versions of Microsoft Word. These, in turn, depict changes in the algorithm over time and highlight the historical significance of the attack, as it has been possible in documents over the range of at least a decade.

Real Evaluation. The present author also evaluated the extent of information leaks in real documents, using the following corpora, which were attained using a combination of webscraping and redaction location algorithms. The latter were custom-made, and included separate algorithms for excising and nonexcising redactions. Both were constructed to identify single-color boxes in-between and on top of text, respectively.

1. *FOIA*. Documents obtained via the US Freedom of Information Act (FOIA) on governmentattic.org [58]. 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 [59]. The OIG is a

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

Table 4: Top: Glyph shifting schemes identified in redacted corpora pages. Bottom: Deredaction results for names tagged.

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 [60]. 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 [61], 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 in on the entire RECAP corpus would be both computationally and financially prohibitive.

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

In this evaluation it was necessary to correctly infer the specific glyph shifting scheme of each document and whether that document was vulnerable, thus a framework for classifying a redaction was developed. This framework included the additional constraint that the redaction could be reasonably classified as a name, in order to reduce the false positives rate created by attempting deredaction on entries outside the evaluated dictionary, FN$\times$LN. A real-world redaction was evaluated if either:

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.

The resulting evaluation broke hundreds of redactions in real world documents. The results of this are reported in Table 4. It is important to note the definition of *broke*: the FN$\times$LN dictionary contained 1.5×10^{10} entries, so a reduction to a few hundred or thousands possibilities is significant. We confirmed with affected parties that often the feasible dictionary is much smaller, and the results for *FNLN* in Table 3 emphasize the attack is dangerous. Many additional verifications of these findings were performed but are omitted from this proposal due to space constraints.

3.2.3 Discussion

Due to the complex technical matter of deredaction and methodological novelties in the evaluation of deredaction attacks, the original publication did not include a description of technical facets of modeling glyph shifting schemes that are essential to correct deredaction. The extraction of glyph shiftng schemes from Microsoft Work, Adobe Acrobat, and other systems was non-trivial and the present author was forced to perform this modeling and reverse engineering largely manually.

One important feature of these algorithms was their dependence on floating point error: they could not be lifted to continuous equations without resulting in an incorrect representation. This is deeply related to the assumptions made in the InteGreat work, which ignores floating point error when lifting continuous control equations from binary firmware for the purposes of verification. Clearly there are domains wherein this floating point error is and is not *useful*, and the proposed dissertation will address this problem in a greater depth by contrasting the two works, something the works, independently considered, could not discuss.

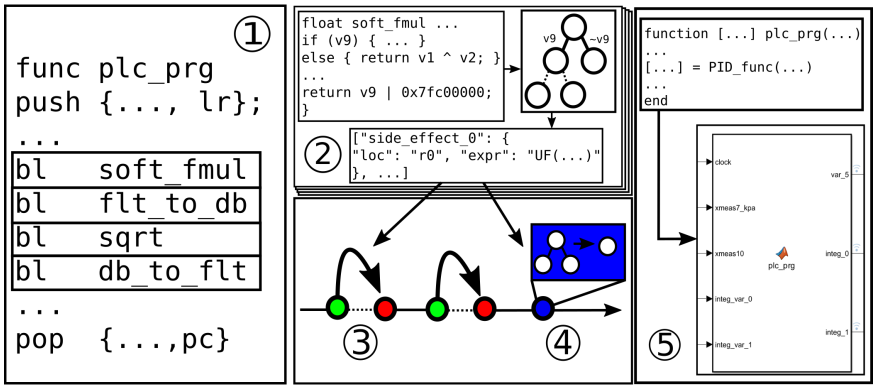
The other key success (and limitation) of *Story Beyond the Eye* was the paper's treatment of the unknown: deredaction may never be considered as determining the correct redacted text, but only as ruling out possibilities with the assumption that the observed evidence is complete, correct, and correctly understood. The paper also highlights important notions core to the use of leaked or remaining information after information destruction (redaction) occurs, most notably the notion of a *prior*, provided by a dictionary. These ideas are, to the present author's knowledge, hardly discussed in the context of lifting and binary program analysis, and both the Jetset and InteGreat works would have benefitted from a richer comprehension of this facet of information loss. The proposed dissertation will more closely consider the role that prior information may play in the domains of firmware rehosting and continuous equation lifting.

3.3 InteGreat

InteGreat reinterprets bitvector-domain symbolic execution of program slices into the theory of uninterpreted functions to perform modular, nestable function summarization and lifting. The tool provides researchers with a novel design language for automatically abstracting complex programs into mathematical equations. In the submitted publication, InteGreat's lifting was used to determine the sensor inputs necessary to precisely destabilize the reactor pressure of a chemical plant, discover novel problems in the domain of firmware rehosting, and discover a flaw in the implemented version of a published quad-copter stabilization algorithm.

3.3.1 Motivating Example

To introduce InteGreat's core contributions, compare the typical operation of symbolic execution on a function in a closed-source binary firmware to InteGreat's approach.



```
func plc_prg
push {..., lr};
...
bl   soft_fmul
bl   flt_to_db
bl   sqrt
bl   db_to_flt
...
pop  {...,pc}
```

```
float soft_fmul ...
if (v9) { ... }
else { return v1 ^ v2; }
...
return v9 | 0x7fc00000;
}
```

```
["side_effect_0": {
"loc": "r0", "expr": "UF(...)"
}, ...]
```

```
function [...] plc_prg(...)
...
[...] = PID_func(...)
...
end
```

Figure 4: Depiction of an example execution of InteGreat on a small program. Red/Green nodes describe abstracting a program slice. Blue stands for program slice pattern triggering higher level abstractions that stitches slices according to combination rules

Traditional Symbolic Execution. Traditional symbolic execution of the function presented in (1) of Fig. 4 will begin by loading a symbolic state *S* at the entry point to the function. Between each inner call, a number of additional instructions could exist but are not depicted. Upon reaching a child function, symbolic execution will enter this function and begin to execute it. The symbolic executor will then proceed to accumulate all the branches, constraints, and operations in the child function. A software floating point multiply we studied, for example, had 153 lines of pseudo-C, 116 of decompiled assembly, and 15 branches. The symbolic executor will then exit the child function and begin executing the succeeding instruction with some *n* possible symbolic constraint sets. This process will continue until the end of the program is reached. Finally, in most cases, the symbol executor will output a statement over the theory of bitvectors rather than a more abstract domain, due to the complexity of determining relevant or desired abstraction boundaries in the general case.

Complexity arises due to the lack of a general solution to the halting problem and the necessity of supporting various microarchitectural operational semantics. A loop without an explicit guard condition within a child function may entirely sabotage symbolic execution due to *state explosion*. For example, a while-true loop may have a break condition which could fire on any of $1\ldots\infty$ iterations (this is common during reads from hardware devices). Moreover, angr [62] and to our knowledge most other closed-source binary symbolic executors do not (publicly) support memory accesses to a symbolic address, as these may potentially reference the entire address space, leading to a combinatorial explosion of the possible resulting execution states. This issue was solved by Coppa et al. [63], however, it was never integrated into angr [64].

InteGreat's Approach. Unlike a traditional symbolic executor, InteGreat adopts a more constrained goal of lifting to continuous equations that can be loaded into a mathematical verifier or program like Matlab. To achieve this, InteGreat provides users with a mechanism for associating program slices to symbols, and handles all the semantic "stitching" this entails automatically. We assume the control flow of a given slice is completely contained within another and there may be no partial overlaps (an independence system). A simple example of this would be the address range of a function compiled inline into the body of another function, or a call instruction itself. The largest slice is the program itself, and slices consist of entry and exit addresses.

First, in (1), we begin a post-order tree traversal of the program slices, descending to "leaf" program slices that have no inner slices. For simplicity, in Fig. 4 we define the slices as each inner function, called to by each *bl* assembly instruction. These slices are provided to InteGreat via a JSON specification file.

Then, in (2), we generate an *abstraction specification* for each given slice. We use traditional symbolic execution to identify all the possible side-effect locations (registers, memory) of the slice for each execution path, which includes function input locations. Note that here we have assumed the desired semantics for the slice's computation are decidable given some *search strategy*.

To avoid state explosion when lifting undecidable algorithms, we leave this strategy user-configurable but provide a few sensible defaults, e.g. taking the state that generates the most complex set of constraints or the largest number of writes to memory. However, state explosion remains a hard problem for abstract interpretation and InteGreat helps to make progress on this problem by providing a more general interface for abstracting program slices. To cope with the problem of combinatorial pointer value explosion discussed above, we implement a version of the 2017 Coppa et al. work, representing each read or write to memory with both the value and the symbolic expression used to determine the address, discussed below.

Next, in (3), we begin the process of *abstraction resolution*. Assuming all nth layer slices are correctly lifted and the symbolic execution state is loaded at the beginning of each slice in the $n+1$th layer. Symbolic execution proceeds as normal until it hits the entry point of an nth layer program slice γ. In the simplest case, the symbolic execution jumps to the exit point of this slice and parameterized uninterpreted function(s) f are written to the output location(s) of γ. To parameterize f, the constraints and execution context at the input locations of γ (locations are identified in (2)) are supplied as parameters to f. f does not have to be an uninterpreted function, but could be any (recovered) model of the child slice's semantics. (3) continues until a higher level abstraction is triggered or the function ends (e.g. return).

(4) represents how each exit point of a program slice triggers a check for patterns specified as higher level abstractions. Upon each pattern match, a corresponding rewrite rule is applied to the abstract symbol tree (AST) containing the symbolic constraints for the current program execution path. This allows us, for instance, to automatically abstract out the marshalling of bitvectors to and from register pairs necessary to represent 64-bit floating point values on 32-bit architectures.

In (5) we post-process the resultant AST, linking output expressions to inputs and resolving abstractions into a model in a high-level language. In this paper, we target Matlab code as it provides an easy, reproducible evaluation. At this stage, program slices are ordered by symbolic execution timestamps, and we disambiguate different pointer values passed into the same program slice using the strict equality discussed above. Finally, we also parse side-effects into stateful variables, i.e. self-loops from output to input. We also remove duplicate computations (AST subtrees) across multiple outputs by translating the AST into a sequence of assignments.

3.3.2 Techniques

InteGreat reformulates classical function summarization (introduced in Sec. 2.3) as *nested function summarization*. A nested function summarization is of the form $\Gamma_1, \Gamma_2 \ \ldots \ \Gamma_n$, where every Γ is a non-empty finite set of dynamic logic formulas. Nested function summarizations can be reused and *chained* inside of higher-level summarizations. The decision procedure for such resolutions follows from Γ.

In order to define a nested function summarization Γ in domain $\mathscr{D}$, we split Γ into two sets, Γ_M and Γ_S, representing the *storage* and *semantics* (expressions) of a program slice, respectively. Given $m \in M$, $\Gamma_M \vdash m_{\mathscr{M}} \to m'_{\mathscr{D}}$, binding the symbols for registers and memory to the abstract domain, and $\Gamma_S \vdash S^{\Gamma}_{\mathscr{M}} \to S'_{\mathscr{D}}$, where $S^{\Gamma}_{\mathscr{M}} \subseteq S_{\mathscr{M}}$, a partial mapping of the semantics of the concrete domain to equivalent semantics in the abstract domain. Partial specification allows users to avoid, for example, reproducing the microarchitecture. A given usecase may only require semantics indicated by `call` instructions. Γ may then be specified for new domains $\mathscr{D}, \mathscr{D}', \ldots$ by substituting $\mathscr{M}$ for the higher layer of abstraction.

```
"pid": {"addrs": [<auto-populated, e.g. 0xCAFE:0xCBFE], "conf": "./pid.json"}
"derivative": {"addrs": [<auto-populated>], "conf": "./derivative.json"}
"integral": {"addrs": [<auto-populated>], "conf": "./integral.json"}
```

```
"in":    {"i0":{"type": "float", "location": "reg", "ptr": "r0"},
          "i1":{"type": "float", "location": "reg", "ptr": "r1"}},
"out":   {"o0":{"type": "float", "location": "reg", "ptr": "r0"},
"expr": "(declare-fun F((_FloatingPoint 8 24)) Bool) ...
  (declare-funinput()(_BitVec32))(...)...
    assert(let((a!1(...(_ to_fp 8 24) i1)..."
```

```
"slice_expr": "f2d, muldf, trunc",
"abst_in":     {"f2d":["i0"], "muldf":["i2", "i3"]},
"abst_out":    {"trunc":["o0"]},
```

Figure 5: Top: Example of a top-level configuration file, with symbolic execution specifications connected to a list of inference rules. **Middle**: InteGreat automatically exports this function summarization of a derivative function as a Z3 function expression with bindings to the input locations. **Bottom**: Example higher-order inference rules with references to nested program slices, used to rewrite multiplication function with type conversion to plain multiplication.

We define program slices as abstraction boundaries that occur naturally in the firmware or by user definition. Each abstraction or summarization is a decision to associate a *sign* (a fresh symbolic variable) with a specific set of computations. The set of substituted instructions, i.e. the range of summarization, must be *complete*: all the program semantics which should be substituted for f must be substituted ($S^f_{\mathscr{M}} \subseteq S^{\Gamma}_{\mathscr{M}}$). This, for example, could require resolving the callsites indicated by dynamically assigned pointer values, something that is hard to do generally, as these values could be I/O dependent: InteGreat addresses this problem by requiring the abstraction of program slices invoking the virtual call (otherwise it will throw a fault). Although in general this problem is also solvable using other means, e.g. symbolically executing more of the program or by using static analysis.

As a computation S can have loops, we must ensure or assume that the nested function summarization Γ is *closed*, i.e. the set of operations $S^{\Gamma}_{\mathscr{M}}$ is finite. The basic solution in cases where S halts for all inputs is universal quantification or the computation of a *fixed point*. For example, let $T = \{t_1, t_2, \dots, t_n\}$ be the complete set of possible traces over computations $C = c_1, c_2, \dots, c_n$ in the program slice. Then the summarization Γ is closed if $\Gamma \vdash \bigwedge_{t \in T} \Gamma_S \vdash t_{\mathscr{M}} \to t'_{\mathscr{D}}$. To allow analysis of $S^C_{\mathscr{M}} \notin HALT$, we use a *search strategy*. In prior work the search strategy is a bound v for unrolling all loops in S (Sec. 2.3).

Nested Abstraction Resolution. Once the particular side-effects of a program slice are *bound*, the side-effect values upon exiting the slice may be substituted with arbitrary logical operations over the program's state, similar to a standard SimProcedure in angr. Unlike angr, InteGreat automates the nesting of procedures based upon a formula, so that more complex abstractions may be built from simpler ones. Representing a function f, we construct $\Gamma \vdash S^f_{\mathscr{M}} \to S^f_{\mathscr{D}}$, but the concrete domains $\mathscr{M}$ includes abstractions previously defined by Γ and $\mathscr{D}$ is not itself "hard-coded" but defined by the logical formulae used to substitute the program slices (either automatically lifted uninterpreted functions or user-specified alternative formulae for the slices' semantics). However, any concrete implementation must restrict $\mathscr{D}$ to some finite set of supported syntaxes, e.g. QF_UF. InteGreat supports the same theories as Z3.

Input. The input to InteGreat is a firmware F and a set of program slices. Slices can 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 abstraction in multiple places, we break this up into a set of objects, where the "entrypoint" object refers to the highest level of abstraction, and descendants refer to inner slices.

The top of Figure 5 provides the list of program slices. *These slices and the JSON file hierarchy can be automatically generated from static analysis of F, and InteGreat provides a set of scripts covering basic use cases, e.g. abstract every child function of a given function.* Thus, in the figure we list the addresses for the slices as "auto-populated". However, we leave the decision to use auto-population scripts to the user, to avoid assumptions about F's structure and the desired abstractions.

As InteGreat executes, each JSON file is populated with the inference rules ($\mathscr{I}$, $\mathscr{A}$) for lifting the program slice. A simplified version of these outputs are given in the middle of Figure 5. Importantly, it contains three parts: the input variables supplied to the inference, the side-effect locations (outputs), and the expression for the slice reinterpreted as an uninterpreted function. Optionally, because this expression is a logical formula over the program state, the user can *modify* this file after the fact and specify alternative or additional formulae for abstracting the program slice.

Finally, once inference rules $(\mathscr{I}, \mathscr{A})$ are generated for sub-slices, we allow these abstractions to be composed at runtime. An example of this is shown in the bottom of Figure 5, which is used to tell InteGreat to compose the given abstractions into a single multiply over the reals, rather than an operation over bitvectors with type conversions.

Output. The output of InteGreat is a vastly simplified Z3 AST composed mainly of uninterpreted functions. The structure of the program slices is decided such that translating the recovered control equations to MATLAB or LaTeX is trivial, and InteGreat includes scripts for this purpose.

Algorithm 1 Abstraction Lifting

```
Input: S, $\mathscr{E}$
Output: $\mathscr{E}'$
 1: ...                                        ▷ Symbolically execute slice $S$, recovering $m_{in}, m_{out}$
 2: for each i in $m_{in}$ do                   ▷ Bind concrete input to abstract expressions
 3:     $m \leftarrow \mathscr{M}(i)$
 4:     $\mathscr{B}_{in}(m) \leftarrow MEM\_READ(m)$   ▷ query InteGreat's symbolic memory model
 5: end for
 6: for each o in $m_{out}$ do                  ▷ Bind abstract expressions to concrete output
 7:     $m \leftarrow \mathscr{M}(o)$
 8:     $e \leftarrow \mathscr{E}(o)$
 9:     $\mathscr{B}_{out}(m) \leftarrow e$
10: end for
11: for each b in $\mathscr{B}_{out}$ do
12:     $v \leftarrow NEW\_ABST()$
13:     $MEM\_WRITE(b, v)$                      ▷ Write abstraction to angr's symbolic memory
14:     $\mathscr{T}(v) \leftarrow TIMESTAMP$
15: end for
16: $\mathscr{E}' \leftarrow BIND(\mathscr{E}, \mathscr{B}_{in}, \mathscr{B}_{out})$   ▷ Bind inference result expressions to outputs
17: $\mathscr{E}' \leftarrow SORT(\mathscr{E}', \mathscr{T})$   ▷ Sort by timestamp
18: for each o in $\mathscr{B}_{out}$ do
19:     $\mathscr{E}' \leftarrow APPLY(\Gamma, o, \mathscr{E}')$   ▷ Recursively substitute abstractions
20: end for
```

Abstraction Lifting Algorithm. Algorithm 1 handles lifting from symbolic variables in $\mathscr{M}$ to $\mathscr{D}$. Input to this algorithm is the set of expressions $\mathscr{E}$, the current recovered abstract semantics of the program in $\mathscr{D}$ and inference rules $\mathscr{I}$ (by default, for lifting to a single uninterpreted function), and the desired program slice to abstract, S. The output is a new state $\mathscr{E}'$ with bindings $\mathscr{B}$ applied to InteGreat's implementation of Symbolic Memory.

InteGreat first retrieves a set of registers and memory locations $m_{\{in,out\}} \in \mathscr{M}$ describing the microarchitectural 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.

InteGreat then dereferences the input locations to the correct symbolic expressions representing their values in InteGreat's symbolic memory model (the inputs may be sets of previously lifted abstractions), and *binds* them into the inputs of the desired abstraction $\mathscr{E}$. InteGreat replaces the existing lower-level, concrete expressions returned by symbolic execution for the slice's outputs with the higher higher-level abstraction via the binding $\mathscr{B}_{out}$.

The last step establishes the lifting results as fresh abstractions to the symbolic memory state. The abstraction is maintained by linking $\mathscr{B}$ into a new lifted program state $\mathscr{E}'$. All future symbolic execution and calls to Alg. 1, will operate on the abstractions provided by $\mathscr{B}_{out}$ rather than the original concrete symbolic expression for each m_{out}.

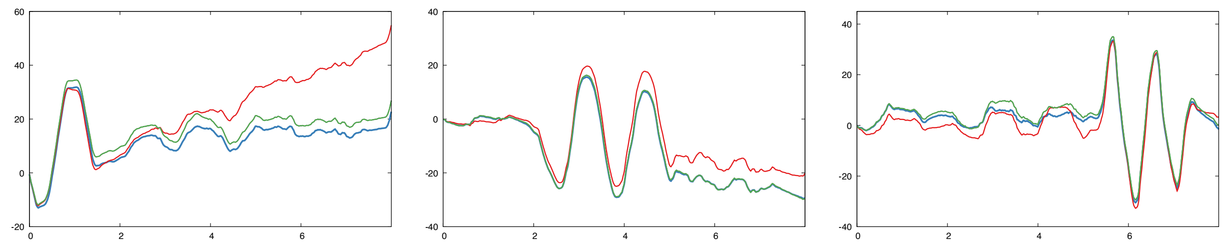
Consider, for example, a floating point multiply over registers r_0 and r_1, with output in r_0. A set of operations are recovered for symbolic inputs i_0 and i_1, yielding the symbolic output $o_0 = i_0 * i_1$. A fresh symbolic variable $abst\_mul\_o0$ would be created and written to the symbolic memory model at concrete register location r_0, and the output of this algorithm would be $\mathscr{E} \cup r_0 = new\_sym \mapsto abst\_mul\_o0(i_0, i_1)$ rather than $\mathscr{E} \cup r_0 = i_0 * i_1$.

The last step of Algorithm 1, lines 17 through 20, is a backward chaining inference algorithm applied on the final expression sets in the derived AST. The lifted program state $\mathscr{E}'$ and a set of expressions for lifting other abstractions described as lambda functions $\Gamma_S \vdash S^{\Gamma}_{\mathscr{M}} \rightarrow S'_{\mathscr{D}}$ in user configuration files (see "slice_expr" in Fig. 5) are applied to resolve output locations into higher-order abstractions. In this process, InteGreat leverages timestamps to ensure that the abstractions are resolved in the correct order.

Symbolic Memory. An important part of the InteGreat workflow is describing input and output locations for each abstraction to obtain symbolic expression trees. 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. Thus it is necessary for InteGreat to allow symbolic expressions as memory addresses.

InteGreat accomplishes this by implementing a symbolic memory model, extending Trtík et al.'s model [65], under-constrained memory [66], and incorporating the core ideas of Coppa et al. [63]. Leveraging the angr event hook infrastructure, InteGreat intercepts all program state modifications, and replaces them with InteGreat's symbolic memory model. During each pointer dereference, the symbolic memory model is queried with the intercepted dynamic state of Symbolic Execution.

Memory Aliasing implemented by proving the equivalence of expressions for pointer addresses. When resolving memory, we use the timestamps of each memory operation and a strict equivalence check on the pointer expression to determine when a memory location is modified. So, for example if $p_1 = 50$ and $p_2 = 50$, and we are checking $read(p_1) = write(p_2)$, then 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 [67].



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

Figure 6: Experimental Results: Orientation Estimation 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.

3.3.3 Evaluation

We evaluated InteGreat by recovering equations for continuous orientation estimation from an open-source, MCU-based quad-copter autopilot controller [68]. The firmware we studied implemented Madgwick's Gradient Descent Orientation Filter [69]. Madgwick provided a C-code implementation of this algorithm in the cited publication.

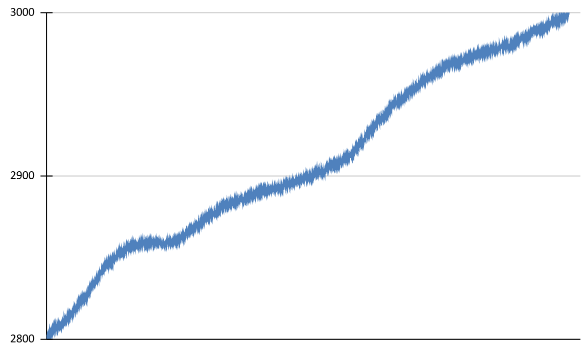
We also extracted continuous equation models from a PLC controlling a pressure-controlling valve in the Tenessee Eastman Challenge [70], a benchmark for modeling and analyzing the dynamics of industrial control systems. The firmware for this PLC was provided by the ICSREF [71] paper, an security paper 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 [72].

Quadcopter 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, through a simulation of the quad-copter spinning in three dimensions. These simulations were performed using Matlab's "rpy_9axis" sensor data. InteGreat's recovered equations allowed us to detect a missing constant multiply and incorporation of Gyroscopic error values in the implemented version of the firmware, as opposed to the published version, demonstrating the utility of the tool. While we do not give details on the precise differences due to space constraints, these are present in the original publication and will be discussed within the proposed dissertation.

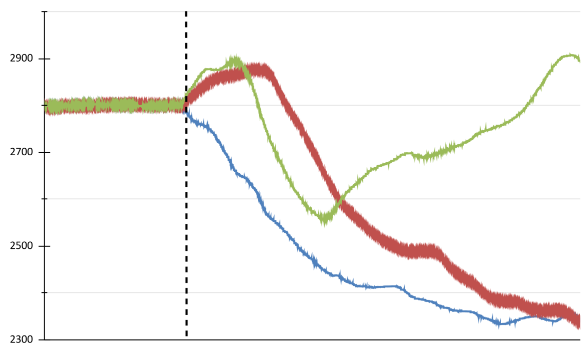
PLC The PLC firmware is written for a WAGO 750-881 in the IEC 61131-3 programming language. The attack in question flipped the proportional gain constant in the first of the firmware's PID calls. By recovering the continuous equations the firmware image implemented, we were able to *stage* this attack without access to the physical PLC. However, we also found an obscure scaling was applied in the firmware to the sensor value for the input reactor pressure:

$$P = 1000(((P_{digital}/30000) - 0.0046)/0.9876) + 2000 \tag{1}$$

This was because the actual reactor pressure (which starts at 2800 kPa in the TE simulation), is supplied to the PLC through a *physical wire*, rather than a serial port. When the Matlab model supplied inputs to the PLC, they were passed through a Digital-to-Analog Converter (DAC). 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.



(a) Naive Attack Reproduction



(b) Voltage Corrected Destabilization

Figure 7: Left: An attempt to stage the ICSREF attack using the recovered model without taking into account the physical effects of digital-to-analog conversion. **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).

We confirmed this was the case with the ICSREF authors and then proceeded to evaluate our findings on the Matlab environmental model of the chemical plant used in the ICSREF paper. The scaling equations applied to PLC I/O values in Matlab are *not* the inverse of Eqn. 1. No scaling was applied to the output in the Firmware's code: the value is simply written to the line. However, on the Matlab side, a conversion was performed to interpret this signal from the voltage level.

Therefore, a naive digital twin of the firmware's implementation, maintaining the original voltage conversions, results in Fig. 7a when staging the attack. This is because, fundamentally, 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. Code for this scaling does not exist in the firmware. This proposes the importance of a hybrid approach to modeling in the current domain of firmware rehosting [73, 74, 75]. 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 new understanding, we were able to correctly reproduce the destabilizing effects on reactor pressure created by uploading code to the PLC (Fig. 7b). We corrected for the input value scaling in all cases, but we explore the effects of *adding* semantics for the output scaling created by physical effects. **Blue** represents the case where the output voltage is exactly matched to the inputs and outputs of Matlab. **Red** represents the case where the output voltage is modified to match the physical scaling created by the physical wiring of the PLC. **Green** represents the case where the attack *did not* occur, and the output voltage scaling is applied to match the dynamics of the physical PLC. These findings explain the otherwise unexplained "bump" in the graph of the original ICSREF paper, and demonstrate how InteGreat can be used to recover an abstract model of a program where no prior model exists.

3.3.4 Discussion

Informed by both the Jetset and *Story Beyond the Eye* 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 abstraction (program slices), and then automates the construction of this abstraction. While this alleviates the 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.

However, the abstraction of complex program semantics is also InteGreat's strength. By 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. Moreover, by incorporating an object-oriented approach for abstraction specifications, the framework is also able to continually expand its base of knowledge. The proposed dissertation will detail the trade-offs of abstraction and discuss the conditions under which InteGreat could be applied to the problems presented in Jetset and *Story Beyond the Eye* thoroughly.

4 Conclusion

These three works, Jetset, *Story Beyond the Eye*, and InteGreat, present three novel perspectives on the recovery of models from binary code. In the first, the models were expression sets corresponding to the constraints of hardware peripheral interactions. The second involved the more complex extraction and hand-modeling of glyph shifting schemes used to determine the information leaked by redacted text. Finally, the third presented a more formal and sophisticated model for the extraction of meaningful abstractions from symbol-stripped, binary code.

As a whole, these works indicate a number of hard problems in the domain of binary program analysis, including the resolution and modeling of precise, discrete floating point operations and their relationship to the real domain, the ability to avoid difficulties incurred in developing a general solution to the halting problem through the use of abstraction, and the impossibility of perfectly rehosting a firmware image into an emulator and accurately interacting with it without *some* knowledge of physical factors and additional hardware that plays a role in the system.

However, the resulting publications also make significant strides towards identifying the proper responses to each of these problems through the use of abstraction, careful evaluation of the algorithms under study, consideration of microarchitectural semantics, and novel binary analysis techniques. The proposed dissertation will work to unify these findings and present them within a framework that is at once both detailed, empirically justified, clear, and unified. It will also work to outline the core limitations of these works, and therefore provide useful commentary for future work in the domain of binary program analysis.

5 Time Line

As the present author has present commitments for employment regardless of the outcome of his doctoral work, the dissertation will be subject to practical constraints on time in accordance with the current University of Illinois policies on dissertation timelines. The absolute minimum time between the preliminary exam proposal and defense is four months, and the current scheduled date of the exam is February 24th. As a result of aforementioned employment constraints, the present author intends to complete the writing of the defense by July 2023 and the proposed defense will occur at the end of this same month, the schedules of the defense committee permitting and considering all other university requirements have been fulfilled.

6 Scientific Merit

7 Broader Impacts

This project will have direct impacts on research and education through access to simulation data products, student training, and K-12 outreach.

Data Access: Maybe write about you will make data available.

Student Training: Write about how you will train students.

Some Other Outreach: Write about more outreach.

Dissemination: Write about how you will disseminate results (i.e., journal articles, workshops, etc).

References Cited

[1] Johnson, E. *et al.* Jetset: Targeted Firmware Rehosting for Embedded Systems. In *USENIX Security Symposium*, 321–338 (2021).

[2] Bellard, F. QEMU, a Fast and Portable Dynamic Translator. In *USENIX annual technical conference, FREENIX Track*, vol. 41, 46 (Califor-nia, USA, 2005).

[3] Schulz, S. & Bockisch, C. A Blast from the Past: Online Time-Travel Debugging with BITE. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, 1–13 (2018).

[4] Wang, F. & Shoshitaishvili, Y. Angr-the Next Generation of Binary Analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, 8–9 (IEEE, 2017).

[5] Zhu, X., Wen, S., Camtepe, S. & Xiang, Y. Fuzzing: a Survey for Roadmap. *ACM Computing Surveys (CSUR)* **54**, 1–36 (2022).

[6] Wenzl, M., Merzdovnik, G., Ullrich, J. & Weippl, E. From Hack to Elaborate Technique—a Survey on Binary Rewriting. *ACM Computing Surveys (CSUR)* **52**, 1–37 (2019).

[7] De Moura, L. & Bjørner, N. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*, 337–340 (Springer, 2008).

[8] Alt, L. *et al.* HiFrog: SMT-Based Function Summarization for Software Verification. 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 II 23*, 207–213 (Springer, 2017).

[9] Milburn, A., Timmers, N., Wiersma, N., Pareja, R. & Cordoba, S. There Will be Glitches: Extracting and Analyzing Automotive Firmware Efficiently. *Black Hat USA* (2018).

[10] Eagle, C. & Nance, K. *The Ghidra Book: The Definitive Guide* (no starch press, 2020).

[11] Cifuentes, C. & Gough, K. J. Decompilation of Binary Programs. *Software: Practice and Experience* **25**, 811–829 (1995).

[12] Chen, Z., Pan, B. & Sun, Y. A Survey of Software Reverse Engineering Applications. In *Artificial Intelligence and Security: 5th International Conference, ICAIS 2019, New York, NY, USA, July 26–28, 2019, Proceedings, Part IV 5*, 235–245 (Springer, 2019).

[13] Cousot, P. & Cousot, R. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 238–252 (1977).

[14] Schwartz, E. J., Avgerinos, T. & Brumley, D. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *2010 IEEE symposium on Security and privacy*, 317–331 (IEEE, 2010).

[15] Cadar, C., Dunbar, D., Engler, D. R. *et al.* Klee: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, vol. 8, 209–224 (2008).

[16] Ball, T. The Concept of Dynamic Analysis. *ACM SIGSOFT Software Engineering Notes* **24**, 216–234 (1999).

[17] Li, J., Zhao, B. & Zhang, C. Fuzzing: a Survey. *Cybersecurity* **1**, 1–13 (2018).

[18] Davidson, D., Moench, B., Ristenpart, T. & Jha, S. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proceedings of USENIX Security 2013*, 463–478 (2013).

[19] Chen, D. D., Egele, M., Woo, M. & Brumley, D. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Proceedings of NDSS 2016* (2016).

[20] Costin, A., Zaddach, J., Francillon, A. & Balzarotti, D. A Large-Scale Analysis of the Security of Embedded Firmwares. In *Proceedings of USENIX Security 2014*, 95–110 (2014).

[21] Costin, A., Zarras, A. & Francillon, A. 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*, 437–448 (2016).

[22] Gustafson, E. *et al.* Toward the Analysis of Embedded Firmware through Automated Re-hosting. In *Proceedings of RAID 2019*, 135–150 (2019).

[23] Clements, A. A. *et al.* HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *Proceedings of USENIX Security 2020* (2020).

[24] Feng, B., Mera, A. & Lu, L. P^{2}IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *Proceedings of USENIX Security 2020* (2020).

[25] Wright, C., Moeglein, W. A., Bagchi, S., Kulkarni, M. & Clements, A. A. Challenges in firmware re-hosting, emulation, and analysis. *ACM Computing Surveys (CSUR)* **54**, 1–36 (2021).

[26] Whelan, C. & Naccache, D. US Intelligence Exposed as Student Decodes Iraq Memo. *Nature* **429**, 116–116 (2004).

[27] Lopresti, D. & Spitz, A. L. Quantifying Information Leakage in Document Redaction. In *Proceedings of the 1st ACM workshop on Hardcopy document processing*, 63–69 (2004).

[28] Bland, M., Iyer, A. & Levchenko, K. Story Beyond the Eye: Glyph Positions Break PDF Text Redaction. *arXiv preprint arXiv:2206.02285* (2022).

[29] Forrester, J. & Irwin, B. An Investigation into Unintentional Information Leakage through Electronic Publication. *Information Security South Africa* (2005).

[30] Hill, S., Zhou, Z., Saul, L. & Shacham, H. On the (In) Effectiveness of Mosaicing and Blurring as Tools for Document Redaction. *Proceedings on Privacy Enhancing Technologies* **2016**, 403–417 (2016).

[31] Müller, J., Noss, D., Mainka, C., Mladenov, V. & Schwenk, J. Processing Dangerous Paths (NDSS, 2021).

[32] Murdoch, S. & Dornseif, M. Far More Than You Ever Wanted To Tell: Hidden Data in Internet Published Documents. `http://irc.smurfnet.ch/events/CCC/congress/21c3/papers/271%20Hidden%20Data%20in%20Internet%20Published%20Documents.pdf` (2004).

[33] Ash, R. B. *Information theory* (Courier Corporation, 2012).

[34] Zhang, H. & Sakurai, K. A Survey of Software Clone Detection from Security Perspective. *IEEE Access* **9**, 48157–48173 (2021).

[35] Aditya V. Thakur and Thomas W. Reps. A Method for Symbolic Computation of Abstract Operations. In Madhusudan, P. & Seshia, S. A. (eds.) *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, vol. 7358 of *Lecture Notes in Computer Science*, 174–192 (Springer, 2012).

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

[37] Furia, C. A., Meyer, B. & Velder, S. Loop Invariants: Analysis, Classification, and Examples. *ACM Computing Surveys (CSUR)* **46**, 1–51 (2014).

[38] Anand, K. *et al.* A Compiler-Level Intermediate Representation Based Binary Analysis and Rewriting System. In *Proceedings of the 8th ACM European Conference on Computer Systems*, 295–308 (2013).

[39] rev. ng: a Unified Binary Analysis Framework to Recover CFGs and Function Boundaries.

[40] Dinaburg, A. & Ruef, A. Mcsema: Static Translation of x86 Instructions to LLVM. In *ReCon 2014 Conference, Montreal, Canada* (2014).

[41] Bland, M., Chen, A. & Levchenko, K. Verifying Continuous Control Equations in Embedded, Symbol-Stripped Firmware. *In Submission, CAV 2023* (2023).

[42] Gluck, P. R. & Holzmann, G. J. Using SPIN Model Checking for Flight Software Verification. In *Proceedings, IEEE Aerospace Conference*, vol. 1, 1–1 (IEEE, 2002).

[43] Babić, D. & Hu, A. J. Structural Abstraction of Software Verification Conditions. In *International Conference on Computer Aided Verification*, 366–378 (Springer, 2007).

[44] Hernandez, G., Fowze, F., Tian, D. J., Yavuz, T. & Butler, K. R. FirmUSB: Vetting USB Device Firmware Using Domain Informed Symbolic Execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, 2245–2262 (Association for Computing Machinery, New York, NY, USA, 2017).

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

[46] Park, J., Pajic, M., Sokolsky, O. & Lee, I. Automatic Verification of Finite Precision Implementations of Linear Controllers. In Legay, A. & Margaria, T. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*, 153–169 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2017).

[47] Jackson, D. & Waingold, A. Lightweight Extraction of Object Models from Bytecode. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, 194–202 (Association for Computing Machinery, New York, NY, USA, 1999).

[48] Ramalingam, G., Komondoor, R., Field, J. & Sinha, S. Semantics-Based Reverse Engineering of Object-Oriented Data Models. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, 192–201 (Association for Computing Machinery, New York, NY, USA, 2006).

[49] Dwyer, M. B. *et al.* Tool-Supported Program Abstraction for Finite-State Verification. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, 177–187 (IEEE Computer Society, USA, 2001).

[50] Ji, R., Hähnle, R. & Bubel, R. Program Transformation Based on Symbolic Execution and Deduction. In *Proceedings of the 11th International Conference on Software Engineering and Formal Methods - Volume 8137*, SEFM 2013, 289–304 (Springer-Verlag, Berlin, Heidelberg, 2013).

[51] Currie, D. *et al.* Embedded Software Verification Using Symbolic Execution and Uninterpreted Functions. *International Journal of Parallel Programming* **34**, 61–91 (2006).

[52] Poeplau, S. & Francillon, A. SymQEMU: Compilation-Based Symbolic Execution for Binaries. In *NDSS* (2021).

[53] Zheng, Y. *et al.* FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *USENIX Security Symposium*, 1099–1114 (2019).

[54] Sandhaus, E. The New York Times Annotated Corpus. `https://catalog.ldc.upenn.edu/LDC2008T19` (2008).

[55] North Carolina Voter Data. `https://www.ncsbe.gov/results-data/voter-registration-data` (2022).

[56] Ohio Voter Data Download. `https://www6.ohiosos.gov/` (2022).

[57] Washington Voter Registration Database Extract. `https://www.sos.wa.gov/elections/vrdb/extract-requests.aspx` (2022).

[58] The Government Attic (2021).

[59] All Inspector General Reports in one Place (2021).

[60] Digital National Security Archive. `https://nsarchive.gwu.edu/digital-national-security-archive` (2021).

[61] Public Access to Court Electronic Records. `https://pacer.uscourts.gov/` (2021).

[62] Issue: Load Symbolic Memory Value, Instead of a Concrete One. `https://github.com/angr/angr/issues/244` (2016).

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

[64] (WIP) Pulls in Memsight Implementation. `https://github.com/angr/angr/issues/1605` (2019).

[65] Trtík, M. & Strejček, J. Symbolic Memory With Pointers. In *International Symposium on Automated Technology for Verification and Analysis*, 380–395 (Springer, 2014).

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

[67] Hind, M. 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*, 54–61 (2001).

[68] eYSIP 2017 contributors. eYSIP-2017 Quadcopter firmware (2017). `https://github.com/heethesh/eYSIP-2017_Control_and_Algorithms_development_for_Quadcopter`.

[69] Madgwick, S. O., Harrison, A. J. & Vaidyanathan, R. Estimation of IMU and MARG Orientation Using a Gradient Descent Algorithm. In *2011 IEEE international conference on rehabilitation robotics*, 1–7 (IEEE, 2011).

[70] Ricker, N. L. Decentralized Control of the Tennessee Eastman Challenge Process. *Journal of process control* **6**, 205–221 (1996).

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

[72] Keliris, A. *et al.* Machine Learning-based Defense Against Process-aware Attacks on Industrial Control Systems. In *2016 IEEE International Test Conference (ITC)*, 1–10 (IEEE, 2016).

[73] Johnson, E. *et al.* Jetset: Targeted Firmware Rehosting for Embedded Systems. In *30th USENIX Security Symposium (USENIX Security 21)*, 321–338 (2021).

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

[75] Clements, A. A. *et al.* HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, 1201–1218 (2020).

[76] Žižek, S. What Rumsfeld Doesn't Know that He Knows about Abu Ghraib. *In These Times* **21** (2004).

[77] Methavanitpong, P. QEMU: Call a Custom Function from TCG. `https://fulcronz27.wordpress.com/2014/06/09/qemu-call-a-custom-function-from-tcg/` (2014).