Verifying the Verifier

Benjamin Ferrell, Kenneth Miller, Matthew Pettersson, Justin Sahs, Brett Webster
{bjf091000, kenneth.miller, mcp085000, jcs074000, brett.webster}@utdallas.edu

***Abstract*—The x86 instruction set architecture, although widely used, remains critical to cyber security as a technology born in absence to the auspices of proper security review, and therefore ripe with design flaws that represent inherent security vulnerabilities. This paper illustrates the design, implementation, and brief evaluation of a machine-code verifier for the native code rewriting and in-lining system (REINS) [13], with the beginnings of a formal correctness proof written in Coq [2].**

Central to our approach is a model of the x86 architecture [7], developed in Coq, specifically targeting Microsoft Windows executables in the Portable Execution file format.

I. INTRODUCTION

For more than twenty years the x86[1] 32-bit instruction set architecture, that was first introduced with the Intel 80386 processor in 1986, has aided technological and academic progression. Yet, arbitrary x86 executables of untrusted origin continue to linger as possible malicious vectors due to the attributes of its design, such as the capability to execute either code or data, which make it a security hazard.

In response to this, and many other cyber threats, many approaches have been researched in great depth. For instance, a somewhat appealing approach is that of the monolithic Virtual Machines, which can be used to monitor execution. VM implementations, however, are large and burdensome.

Our project leverages a light weight solution, utilizing a software-based fault isolation (SFI) approach, the REINS project. Much success has been made towards securing untrusted code via SFI implementations and providing machine checked proofs of validity for strong confidence such as PittSFIeld [5], NaCl [14] and RockSalt [7]. However these early implementations required the code producers support, such as gcc-produced assembly code, program database files, or debug information. If the implementations respective requirements are not supplied, SFI cannot be enforced by any of these endeavors. This failure simply rejects the native code, and therefore, provides no benefits. The end effect is a severely restricted set of applicable cases, with the complete circumscription of legacy binaries.

REINS [13], the latest in an ever evolving set of SFI systems, is the first compiler-agnostic, machine-certifying, x86 specification and implementation that suports real-world COTS binaries without the need for any developer supports source, debugging symbols or any other additional information other than the executable itself. Our work builds upon the RockSalt x86 formal model in order to implement the REINS algorithm and subsequently a verification of it in Coq.

[1]We use x86, IA-32, x86-32, and i386 interchangeably

In summary, our work makes the following contributions specific to REINS:

- We extend the RockSalt x86 formal model in Coq to support a subset of the Portable Executable (PE) file format (we mainly model the parts of the PE format relevant to the REINS verification algorithm).
- We implement the REINS verification algorithm.
- We lay the foundation for a proof of the security properties that follow from the formal specification.

The remainder of this paper is organized as follows. Section 2 develops essential or key architectural concepts that are then elaborated in section 3, which addresses detail. We then categorically assess the system in section 4. In sections 5 and 6 we discuss relevant prior systems and the future of our work, respectively. Section 7 concludes.

II. BACKGROUND

Due to the large and often perilous nature of the x86 instruction set, a machine checked formal proof of correctness is often desired. As in complex or mission critical systems, preventing vulnerabilities with a high assurance is only feasible or even achievable through formal means.

Hence the goal of our project is to implement a verified Coq checker for REINS rewritten executables. Thus ensuring rewritten binaries provably adhere to REINS correctness properties, and in turn, are free of malicious instructions or unforseen logic errors (bugs).

III. DETAILED DESIGN

In this section we illustrate our architecture and provide a detailed design of our project. Mirroring our progression throughout the semester, our project naturally consists of three subsections:

(a) An augmented RockSalt model, poised as a foundation of this and future work
(b) the addition of portable executable components
(c) our implementation of the REINS algorithm and a machine checked proofs that our implementation follows the REINS formal specification

When combined, these form the beginning of proving inherent security properties that follow from the conjugation of all properties outlined in the REINS formal specification.

A. Building on the RockSalt Model

Morrisett *et al.* greatly advanced NaCl with their contributions of RockSalt. The RockSalt Coq model for the x86 architecture consist of three major stages: (1) a decoder to translate bytes into an abstract syntax, (2) a compiler that translates the

abstract syntax into a sequence of RTL instructions, and (3) an interpreter for the RTL instructions.

The abstract syntax for instruction is generated by the x86 model's decoder. The translation is specified via generic grammars constructed in a domain-specific language embedded in Coq. The language lets users specify a pattern and associated semantic actions for transforming input strings to outputs such as abstract syntax. The language is limited to recognizing regular expressions; however, the semantic actions are arbitrary Coq functions. In RockSalt and our project, these semantic actions are defined using an inductively defined predicate, which makes it easy to symbolically reason about grammars.

While the denotational specification makes it easy to reason about grammars, it cannot be directly executed. Consequently, RockSalt includes a parsing function which, when given a record representing a machine state, fetches bytes from the address specified by the program counter and attempts to match them against the grammar and build the appropriate instruction syntax.

The parsing function is defined by taking the derivation of the x86 grammar, with respect to the sequence of bits in each byte, and then checking to see if the resulting grammar accepts the empty string. Reasoning about derivatives is much easier in Coq than attempting to transform grammars into the usual graph-bases formalisms, due to not needing to worry about issues such as naming nodes, equivalence on graphs, or induction principles for graphs. Calculating the derivative, including the appropriate transformation on the semantic actions are skillfully provided as straight forward functions in the RockSalt paper [7].

Once the iterated derivative of the grammar is calculated with respect to a string of bits, the set of related semantic values are easily extractable by running an extract function, which returns those semantic values associated with the empty string, also in the RockSalt paper [7].

Our efficient REINS checker, derived from a marriage of the NaCl checker and the REINS algorithm, as discussed in the REINS specific section, is built from a deterministic finite-state automaton (DFA) generated off-line, re-using the definitions for the grammars and derivations in the parsing library.

The second stage translates the parsed bytes into a RISC-like register transfer list (RTL) language for bit-vectors. This language abstracts over the x86 architecture's definition of machine states as a finite map from addresses to bytes. For each x86 instruction, the RockSalt model defines a function that translates the abstract syntax into a sequence of RTL instructions. This translation is encapsulated in a monad that addresses allocating fresh local variables, and that allows the construction of higher-level operations from sequences of RTL commands.

Once the decoder and translation to RTL are defined, the next step is to define a semantics of RTL instructions. These semantics are given as small-step operational semantics, encoded as a step function from RTL machine states to RTL machine states. RTL machine states record the value of the various x86 locations, the memory, and the values of the local variables. It is worth noting that one of the RTL instructions is an instruction that fetches an x86 instruction from memory, then decodes and executes it.

B. Portable Executables

Due to the nature of some of the REINS formal specifications, much of the algorithm could not be implemented prior to developing a capability to incorporate a representation of the executable as it resides on disk to Coq. The applicable specifications that stipulated this requirement were:

- executable sections reside in low memory
- no exported symbols target low memory chunk boundaries
- computed jumps through the Import Address Table must be preceded by masking operations

Because Coq lacks I/O, we implemented a short OCaml script that reads an executable file as a list of bytes. This script then calls OCaml code generated by Coq's extraction feature, allowing our program to use the RockSalt parser. In order to be able to prove anything about these mentioned requirements, however, we needed to traverse the Portable Executable file format.

Among the challenges of traversing the PE file format was a severe lack of documentation. It is well known among the security community that the PE header provided by Microsoft isn't entirely accurate (security by obscurity), but that there are small discrepancies. Using the reverse engineering community's resources, we found a document explaining the structure of the PE file format [11]. Even then, traversing the PE file from what you manually read using a hex editor can be rather confusing. To address this critical issue, a well-known program among reverse engineers (CFF explorer) was used in tandem with a hex editor in order to manually perform the traversal just as would be required in Coq.

To instill some confidence in our traversal implementation, we made several examples of regular PE files that could be found on a typical windows machine, such as defragmentation software. Initially receiving errors from manually parsing compared to what the CFF explorer was showing, eventually the problem was traced down to a address translation issue. When an executable is on disk, it contains addresses that represent where a value will be at runtime, which is different from when it is on disk. In order to be able to compensate for the difference, the translation had to be written in Coq. These addresses are called Relative Virtual Addresses, and they are of the form[2]:

$$BaseAddress + RVA - SectionHeaderVA + SectionHeaderPtrToRawData = AddrssInFile$$

Our implementation makes a few minor assumptions due to the fact that validating would have required insider help from Microsoft:

[2]That is, the section header's data in question is actually dependent on the RVA's value.

- that PE's in general conform to specifications closely enough to work
- that dishonest or maligned executables are discarded by the implemented REINS system
- that our address translations are accurate to what translations are performed by real systems.

Aside from these, after having successfully implemented the traversal functions, CFF explorer was used to check what was retrieved using the algorithms with good success.

Lastly, among the steps included in our partial initialization of a PE header in Coq, we traverse the PE header with regards to the Import Address Table by retrieving information from the Optional Header's data directory and the Export Table and executable sections by manual traversal of the PE beyond the Optional Header.

C. The REINS Verification Algorithm

The verification algorithm, as detailed in [13], consist of a fall-through disassembly of each executable section in the binary checking for the following properties:

- All executable sections reside in low memory.
- All exported symbols (including the program entry point) target low memory chunk boundaries.
- No disassembled instruction spans a chunk boundary.
- Static branches target low memory chunk boundaries.
- All computed jump instructions that do not reference the IAT are immediately preceded by the appropriate and-masking instruction.
- Computed jumps that read the IAT access a properly aligned IAT entry, and are preceded by an and-mask of the return address.
- There are no trap instructions.

These syntactic properties ensure that any unaligned instruction sequences concealed within untrusted, executable sections are not reachable at runtime, and that the REINS runtime library cannot be bypassed by manipulating return addresses or computed jump targets. Thus, at runtime, the rewritten executable cannot violate the security properties enforced by the REINS runtime library. We implemented this by modifying RockSalt's implementation of the NaCl verification algorithm, which is structurally similar.

IV. EVALUATION

We tested our implementation against a set of hand tailored binaries to serve as a proof of concept, as we have not had access the REINS rewriter, or any rewritten binaries. The test case executables and expected results corresponding to the various REINS properties, used as a basic sanity check, are shown in Table I.

TABLE I
BINARY EXECUTABLES USED FOR TESTING.

REINS Property Violation	EXE	Result
CALL not at end of chunk boundary	callNotAtEnd	Fail
Disallowed Interrupt	interrupt	Fail
Disallowed System Call	syscall	Fail
Not masking the expected register on JMP	maskOnWrongReg	Fail
Not masking the expected register on CALL	noAndBeforeCall	Fail
Incorrect Mask	wrongMask	Fail
No Violations, No IAT call	safe	Pass
Call through IAT, No Violations	callThruIAT	Pass

V. RELATED WORK

Binary "sandboxing" was first introduced as a technique for fault-isolation by Wahbe *et al.* in [12]. The key contribution of their work was to show that by directing all unsafe operations through a dedicated register, a jump to any instruction in the code region could be safe. However, their approach is not immediately applicable to CISC architectures, like the Intel IA-32, which include variable-length instructions.

McCamant and Morrisett described an SFI technique applicable to CISC architectures as well as two optimizations not previously employed. Their implementation, referred to as the Prototype IA-32 Transformation Tool for Software-based Fault Isolation Enabling Load-time Determinations (of safety), or better known as PittSFIeld [5].

Arguably the most well-known and distributed system is Native Client, a service provided for Google's Chrome browser that allows native executable code to be run directly in the context of the browser [14]. NaCl's contributions include an open source infrastructure for OS and browser-portable sandboxed x86 binary modules, support for advanced performance capabilities such as threads, SSE instructions, compiler intrinsics and hand-coded assembler. Similar to PittSFIeld, NaCl requires untrusted code to be compiled with their specialized tool chain.

Coq-based systems are gaining traction as a means to guarantee desired correctness properties. For example, CompCert is a compiler that generates PowerPC, ARM and x86 assembly code from CompCert C, a large subset of the C programming language. Written mostly in Coq, the CompCert compiler is proven to be correct; that is, the CompCert project leverages the power of Coq to prove that the assembly code generated by the compiler is semantically equivalent to the source program.

VI. FUTURE WORK

Our augmented x86 model is minimal; we do not yet handle floating-point instructions, system programming instructions, nor any of the MMX, SSEn, 3dNow! or IA-64 instructions. In addition, some sequences of pads are not correctly recognized. As section 4 suggest, our restricted model is sufficient for a subset of real executables, albeit a somewhat trivial subset.

Although we have made great strides, a large requriement in moving forward, is a complete x86 model. That is, we wish to extend our x86 model to include all IA-32 instructions such that any binary produced by the REINS rewriter is verifiable by our checker.

We would like a complete verifier correctness proof. Such a proof requires, as prerequisites, many auxiliary lemmas such as a formal proof of PE correctness. We suspect that several other bugs have not yet been discovered, and recognize the need for further, more comprehensive, evaluation of our implementation.

VII. CONCLUSION

We have presented a formal model to serve as a foundation for a formally verified IRM and SFI checker. This includes significant additions to the RockSalt abstract x86 model as well as a implementation, developed in Coq, of the REINS verifier. We hope this work will aid in the expansion of formal proofs of security on the x86 architecture.

ACKNOWLEDGEMENTS

We would like to thank Xavier Leroy [4] and Joseph Tassarotti [10] for the insight they provided on RockSalt.

REFERENCES

[1] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 325462-042us edition, 2012.

[2] Coq development team. The coq proof assistant. http://coq.inria.fr/, 1989-2012.

[3] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[4] Xavier Leroy. http://pauillac.inria.fr/~xleroy/, 2012.

[5] Stephen McCamant and Greg Morrisett. Evaluating sfi for a cisc architecture. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[6] Microsoft. Microsoft portable executable and common object file format specification, 1999.

[7] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. Rocksalt: better, faster, stronger sfi for the x86. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 395–404, New York, NY, USA, 2012. ACM.

[8] GoNative Participants. Gonative safe execution of native code. http://sos.cse.lehigh.edu/gonative/index.html, 2012.

[9] Matt Pietrek. Peering inside the pe: A tour of the win32 portable executable file format. Technical Report Vol.9 No.3, Microsoft Systems Journal, March 1994.

[10] Joseph Tassarotti. http://pauillac.inria.fr/~xleroy/, 2012.

[11] The Collaborative RCE Tool Library Team. Collaborative rce tool library. http://www.openrce.org/reference_library/files/reference/PE\%20Format.pdf, 2012.

[12] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev.*, 27(5):203–216, December 1993.

[13] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, Orlando, Florida, December 2012. forthcoming.

[14] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Orm, Shiki Okasaka, Neha Narula, Nicholas Fullagar, and Google Inc. Native client: A sandbox for portable, untrusted x86 native code. In *In Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2009.