2017 IEEE Symposium on Security and Privacy Workshops

Research Report: Hardware-Enforcement of Walther-Recursive Program Functions

Jacob I. Torrey, Mark P. Bridgman and Tomasz Tuzel
Assured Information Security
Greenwood Village, CO, USA
{*torreyj, bridgmanm, tuzelt*}*@ainfosec.com*

***Abstract*—Preliminary experiment design and research goals are presented to measure the security implications of using just-in-time (JIT) compilation in conjunction with hardware-enforced restrictions on the computational complexity of general purpose applications. LLVM passes are used during compilation to mark sub-components of an application if they are provably halting (Walter recursive). A modified JIT engine is under development to unroll loops based on run-time semantics, while Intel's Processor Tracing technology is used to enforce those execution bounds. This paper describes an ongoing effort to improve security through the realization and enforcement of restricted computational environments. As an example, an analysis of the impact on the attacker when return-oriented programming is limited to unidirectional execution highlights one such computationally-derived security benefit of this work.**

***Keywords*-Walther recursion, programming languages, parsing, LLVM, language-theoretic security, LangSec, processor-tracing, just-in-time compilation.**

I. Introduction

To reduce the risk of software exploitation, Language-theoretic security (LangSec) provides two fundamental explanations of why exploits, or "weird machines" [1] crop up: *ad hoc* parsing of attacker-controlled data interleaving syntactic and semantic analyses, and overly-powerful programming/run-time environments. This work aims to bring a LangSec-inspired reduction of the computational "privilege" present in most current environments to bear with minimal overhaul of the development process.

If programs can be limited in their complexity and power, reference monitors and formal verification can improve the security when exposed to attacker-controlled input. In previous works [2], [3], the authors have shown both empirical evidence of the improvement of verification as well as the broad applicability of such restricted computational environments. Presented herein is the design and initial evaluation of a system that can automatically provide hardware-enforcement of termination for program components identified to execute in a sub-Turing Complete space.

A. Contributions

In [2], the sub-Turing complete programming language *Crema* was shown to ease verification and ensured programmers better provided their intent into implementation, reducing the risk of certain classes of attack [4] [5]. *Moka* [3] determined that these programming design patterns are broadly applicable to general purpose programming, and what classes of software modules require more computational power than others. This paper describes *Ristretto*, a compiler and run-time tool-chain that automatically restricts large portions of program code into a provably-halting environment even with a powerful attacker model exemplified, but not limited to, return-oriented programming [6] (ROP)-type attack where execution flow is redirected. Intel's Processor Tracing [7] functionality is used by a monitor application that interfaces closely with the JIT compiler of the target application to prevent backward branching in unrolled loops and other restricted program modules.

II. Problem Specification

A. Background

The following paragraphs provide a brief background to the concepts built upon in this work.

LLVM and Clang: The LLVM compiler framework [8] is a tool-chain of modular components to analyze, optimize, compile, and execute programs via a standardized byte-code intermediate-representation (IR). Front-ends parse an input language, construct an abstract syntax tree (AST), and emit LLVM IR, which then can leverage the existing optimization passes, a cross-platform JIT engine, and static analysis tools to allow for rapid compiler development. One such front-end parser for the C and C++ programming languages is Clang, which transforms source files into an LLVM AST for generation of IR, bit-code, or binaries to be run natively on a target platform.

Once an input program has been converted to LLVM IR, there are a number of tools and libraries that can then be used to optimize the IR for faster execution or smaller memory footprint. Of interest to us is the *Pass*, a construction for modules that can navigate the AST either for analysis or modifications. Passes provide a powerful API for the program's AST and can specify which AST elements to inspect or modify: viz., loops, functions, modules, etc. Typically, passes are used during compile-time, however, in order to leverage the run-time semantics available to more narrowly bound loop execution, the JIT engine is able to perform transformations just prior to execution. After the run-time modifications, the JIT engine will generate the machine code for that function or module and execute it.

DOI 10.1109/SPW.2017.30

Moka Findings: As presented in [3], we have found that loops that require unknown or absent bounds on termination are exceedingly rare, with such loops highly focused in areas dispatch loops for events, UI and scheduling. Our final findings for the Linux 4.4 kernel (x86) was that only 11.2% of loops in the kernel were *not* provably terminating.[1] Considering the complexity required of a kernel to schedule tasks and handle events, application code may have even fewer of such unbounded loops. In this light, we deemed the usage of the restricted computational environment productive and with a broad applicability.

Intel Processor Tracing: Intel Processor Tracing (PT) [7] is a new execution tracing feature implemented with hardware facilities in the latest Intel CPUs.[2] It is similar to the Branch Trace Store (BTS) mechanism implemented in Pentium 4 and later processors with a few key differences. Notably, Intel PT provides low-overhead capture of program flow making it suitable for production builds of most applications. The trace records are more detailed than BTS and consist of packets that represent different types of execution branches that can be used to reconstruct a complete execution flow.

Return-Oriented Programming: ROP is used commonly in environments where the program stack and data memory is marked as non-executable. This attack is a type of "weird-machine" in which the execution flow through a program is rerouted to express the attacker's semantics instead of the developers. A target application is mined for existing "gadgets" that perform some of the semantic building blocks needed to implement the attacker's shellcode. These gadgets are then chained together by filling the vulnerable program's stack with return addresses pointing to the gadgets (typically ending in a `RET` instruction), allowing the attacker to reuse the existing instructions in a different order. These gadget addresses can be thought of as an assembly language to be executed within the *ad hoc* virtual machine environment emergent from the vulnerable application.

B. LangSec-inspired Challenges

LangSec highlights the risks involved with parsing input into a program's internal type-system and demonstrates how a poorly designed or implemented parser creates a risk of unintended computations. The theory goes further and calls for input languages to be as restricted as possible, urging programmers to utilize the minimum amount of computational expressiveness necessary to validate inputs. This paper specifically examines the feasibility for a compiler tool-chain to automatically incorporate these restricted computational environments into general purpose applications—yielding safer programs.

[1]This included counting all the schedulers when only one is actively executed at a time

[2]Skylake and newer micro-architectures

III. Approach

To identify every loop at run-time as terminating or not is equivalent to the Halting Problem [9] — determining whether or not a general algorithm (equivalent to a Turing Machine) will halt on the given input. The scope of this effort is such that perfect coverage is not expected, instead Ristretto will focus on protecting the regions of a program that are provably sub-Turing complete (i.e., in the Walter Recursion complexity class [10]).

A. Compile-Time Function Identification

Ristretto will rapidly classify a large number of components with an LLVM pass developed for [3] that marks every function as a candidate for run-time protections that either: does not contain any looping construct, or only contains looping constructs that fall within the bounds of Walter recursion — meaning termination is provable. These loops may already be candidates for unrolling during optimization or have fixed bounds. Those loops with defined bounds are marked for JIT unrolling, as in many cases the bounds are `MAX_INT` without additional run-time semantics.

B. JIT Loop Unrolling

During the execution of the protected application, when a region of the program that has been identified as provably bounded is reached a run-time algorithm is used to determine a more narrow bound on loop iterations before unrolling the loop into machine code for that execution. At the boundaries of the this unrolled region of machine code, a synchronization function is embedded to alert the PT monitor that branches to lower addresses (i.e., backwards) should be trapped on.

Within LLVM, this requires the "hoisting" of the loop into a separate function and module in order to prevent the JIT-compiled machine code from being cached and reused on future executions where the loop bounds may be different. In essence, the target loop is moved to an anonymous function in a temporary module, with the references to local and global variables updated to point to the context of the original function. At this point, the run-time loop bounds can be calculated using the *induction variable* and the values at loop entry. As the function identification step in Section III-A only marks loops that are known to terminate within at least `MAX_INT` iterations, the JIT engine will only operate on loops where the run-time information is sufficient to determine the back-edge count.

C. Forward-only Enforcement

A run-time mechanism implemented with Intel PT can be used to identify deviations from expected execution flow and to terminate the offending process. Expected execution flow for many functions is determined by LLVM passes at compile time or with the JIT loop unroller. Unrolled functions can be expected to have a unidirectional execution

flow and thus should not be the source of branches that decrease the instruction pointer. Such functions are framed with calls to an Intel PT library to trace the execution and parse the recorded execution flow.

Return instructions that redirect execution to a caller at a lower address can also be accommodated by saving the expected return address when the trace starts. Asynchronous observation of the trace capture provides an indication of any deviations from expected program flow, triggering process termination.

Any attacks against a function protected by Ristretto must depend on forward-only branches that do not utilize return instructions (or sequences of instructions that perform similar semantics) in order to succeed, limiting the attacker to a weaker class of computational complexity. Ristretto provides the same level of power reduction on the attacker regardless of specific attack class including: ROP, stack-based buffer overflow, or even a code injection attack allowing the attacker to completely rewrite the code of a protected region.

IV. Experiment Setup

Due to the on-going nature of this research report, we are unable to provide detailed results about the execution and performance of the Ristretto system as a whole. In lieu of these results, a study was performed to measure the impact on a ROP attack if the attacker is constrained to execute the gadget chains in a *forward-only* environment—one in which gadget reuse is forbidden. Additionally, a basic performance study on Intel PT is included for a rough projection of the run-time overhead.

In order to evaluate the potential security improvements afforded by the Ristretto environment, an empirical analysis was performed on the impact of forward-only execution on an example attack class: ROP. The tool Ropper [11] was used to find ROP chains in the target binaries to perform selected attacks. Ropper was then modified to only form chains of gadgets in ascending memory order—simulating execution in a forward-only environment. While there are weaknesses to this approach due to the loop unrolling performed in the JIT engine allowing for limited reuse of gadgets, most of the gadgets are selected from the libraries or other program functions and not within the function itself.

A. Results and Discussion

As this is a research report on an on-going effort, the results are not complete. Only the findings of the modified Ropper tool and the Intel PT overhead studies will be described until further development and testing is completed.

1) Ropper Analysis: Ropper was tasked with finding a gadget chain for the following attacks:

- `execve("/bin/sh")` attack to execute a shell within the vulnerable process
- `mprotect` attack to weaken the memory protections for a Linux application to ease exploitation
- `VirtualProtect` attack to weaken the memory protections for a Windows application to ease exploitation

Each of these were tested in both 32-bit and 64-bit modes. For the Linux-based attacks, the following two targets were used:

- Statically-linked `libc` (Appendix B provides an example ROP chain for the `execve` attack against a 64-bit `libc`)
- Dynamically-linked `mysqld` chosen due to its larger application size at between 200-245 MB depending on architecture

The Windows-based attack (`VirtualProtect`) was attempted against the below two targets:

- Microsoft's `cmd.exe`
- Cygwin's `bash.exe`

The initial findings show that in all cases, Ropper is able to find one or more valid ROP chain of gadgets within the target. When Ropper was restricted to only search for chains adhering to the forward-only model, it was *unable* to generate a chain for any attack $\times$ target $\times$ architecture. With this preliminary findings, we are optimistic that ROP-type attacks will be significantly hampered by the Ristretto compile- and run-time protections in addition to other attack classes by fundamentally restricting their computational power.

2) Ristretto Weaknesses: Ristretto aims to operate on unmodified applications that may be designed and developed in such a manner to be vulnerable to a wide range of attacks without changing the underlying program semantics. As such there are notable limitations to the security protections afforded by an automatic compiler and run-time monitor solution. Primarily of note is the lack of protections against attack on program regions with loops that cannot be automatically classified as within the Walther recursion class—a likely vulnerable target due to their complexity. Once an attacker is able to exploit a vulnerability in one of these unprotected regions, there are no checks performed and arbitrary attacks can be executed. The developed test-suite includes many adversarial cases designed to bypass Ristretto, including self-modifying code that changes loop semantics *during* execution, or cross-modifying multi-threaded applications that make enforcement extremely challenging.

Another weakness could be the memory usage and performance of performing JIT unrolling for each loop, especially those with a large number of iterations (e.g., a timer loop). In these scenarios, the impact on execution may be too great and would cause instability.

In order to gain a sense of the performance overhead that the Intel PT monitoring thread will impose on the target application, a CPU-bound test program was developed to compute an approximation of π using an inefficient power-series definition (see Appendix A for source). This program

was run both with and without Intel PT monitoring its branching patterns, but not preventing backward jumps as the loop was not unrolled; the performance impact (Table I) was approximately 5%. Each test was executed ten times with differing numbers of iterations to determine if there is a constant overhead for PT.

During the testing phase of this effort, loops of various sizes will be tested with many large bounds on iterations in order to measure if the performance becomes untenable. In such a situation, a sufficiently large number of unrolled iterations could be divided and the PT monitoring loop apprised to allow a fixed number of backward branches from the end of the unrolled loop to the beginning.

V. Conclusion

The ongoing Ristretto effort aims to measure the feasibility of developing general purpose applications with a closer eye towards the computational complexity of each component. By automatically identifying regions of code that are provably terminating and enforcing that termination bound, it is hypothesized that vulnerabilities could be mitigated. Through the use of a restricted execution model, software safety can be increased to reap the benefits of a LangSec-inspired approach without redesign or development. Future research and development aims to net performance and real-world security metrics for the consideration of the practical deployment of the Ristretto compiler tool-chain in a production environment.

Acknowledgments

This material is based upon work supported by the United States Air Force under Contract No. FA8750-15-C-0133[3].

References

[1] J. Vanegue, "The weird machines in proof-carrying code," in *Proc. First Annual Langsec Workshop*, May 2014.

[2] J. Torrey and M. Bridgman, "Verification state-space reduction through restricted parsing environments," in *Security and Privacy Workshops (SPW), 2015 IEEE*, May 2015, pp. 106–116.

[3] J. I. Torrey and J. Miodownik, "Research report: Analysis of software for restricted computational environment applicability," in *2016 IEEE Security and Privacy Workshops (SPW)*, May 2016, pp. 185–188.

[4] M. Dowd. (2003) Sendmail release notes for the crackaddr vulnerability.

[5] J. Vanegue, S. Heelan, and R. Rolles, "Smt solvers for software security," in *Proceedings of the 6th USENIX Conference on Offensive Technologies*, ser. WOOT'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2372399.2372412

[3]Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force.

[6] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. ACM, 2007.

[7] I. Corporation. (2013) Processor tracing. [Online]. Available: https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing

[8] C. Lattner. (2015) The LLVM compiler infrastructure. [Online]. Available: http://llvm.org/

[9] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[10] C. Walther, "Security applications of formal language theory," *Artificial Intelligence*, vol. 70, no. 1, 1994.

[11] S. Schirra. (2016) Ropper. [Online]. Available: https://github.com/sashs/Ropper

Appendix A.
π Calculation Algorithm

```
int iterations = 0, i;
float result = 0.0;
for (i = 0; i <= iterations; ++i)
    {
      /* Pi approximation: Sum from i->infinity of
           ((-1)^i)/(1+(i*2)) */
      if (0 == (i%2)) // intentionally sloppy to
           slow up calculation
        {
          result += ((float)1)/((float)(1+(i*2)));
               // casts keep the ints away
        }
      else
        {
          result -= ((float)1)/((float)(1+(i*2)));
        }
    }
```

Figure 1. Loop to approximate π with power series

Appendix B.
Example Ropper Output

The below is an example of Ropper's output when it is successfully able to generate a chain of gadgets to perform the attack semantics:

Number of iterations	Real time without tracing (mean)	Real time with tracing (mean)	% overhead
900,000,000	5.2523s	5.5164s	5.02%
1,800,000,000	10.5108s	11.0167s	4.81%

Table I
PERFORMANCE IMPACT OF INTEL PT MONITORING

```
rop += rebase_0(0x0000000000001496) # 0
    x0000000000401496: pop rdi; ret;
rop += '//bin/sh'
rop += rebase_0(0x00000000000bd134) # 0
    x00000000004bd134: pop rax; ret;
rop += rebase_0(0x00000000000ca080)
rop += rebase_0(0x000000000007a344) # 0
    x000000000047a344: mov qword ptr [rax], rdi;
    pop rbx; ret;
rop += p(0xdeadbeefdeadbeef)
rop += rebase_0(0x0000000000001496) # 0
    x0000000000401496: pop rdi; ret;
rop += p(0x0000000000000000)
rop += rebase_0(0x00000000000bd134) # 0
    x00000000004bd134: pop rax; ret;
rop += rebase_0(0x00000000000ca088)
rop += rebase_0(0x000000000007a344) # 0
    x000000000047a344: mov qword ptr [rax], rdi;
    pop rbx; ret;
rop += p(0xdeadbeefdeadbeef)
rop += rebase_0(0x0000000000001496) # 0
    x0000000000401496: pop rdi; ret;
rop += rebase_0(0x00000000000ca080)
rop += rebase_0(0x00000000000015b7) # 0
    x00000000004015b7: pop rsi; ret;
rop += rebase_0(0x00000000000ca088)
rop += rebase_0(0x0000000000041c36) # 0
    x0000000000441c36: pop rdx; ret;
rop += rebase_0(0x00000000000ca088)
rop += rebase_0(0x00000000000bd134) # 0
    x00000000004bd134: pop rax; ret;
rop += p(0x000000000000003b)
rop += rebase_0(0x0000000000066605) # 0
    x0000000000466605: syscall; ret;
```

Figure 2. Ropper output for `execve` attack against a 64-bit libc