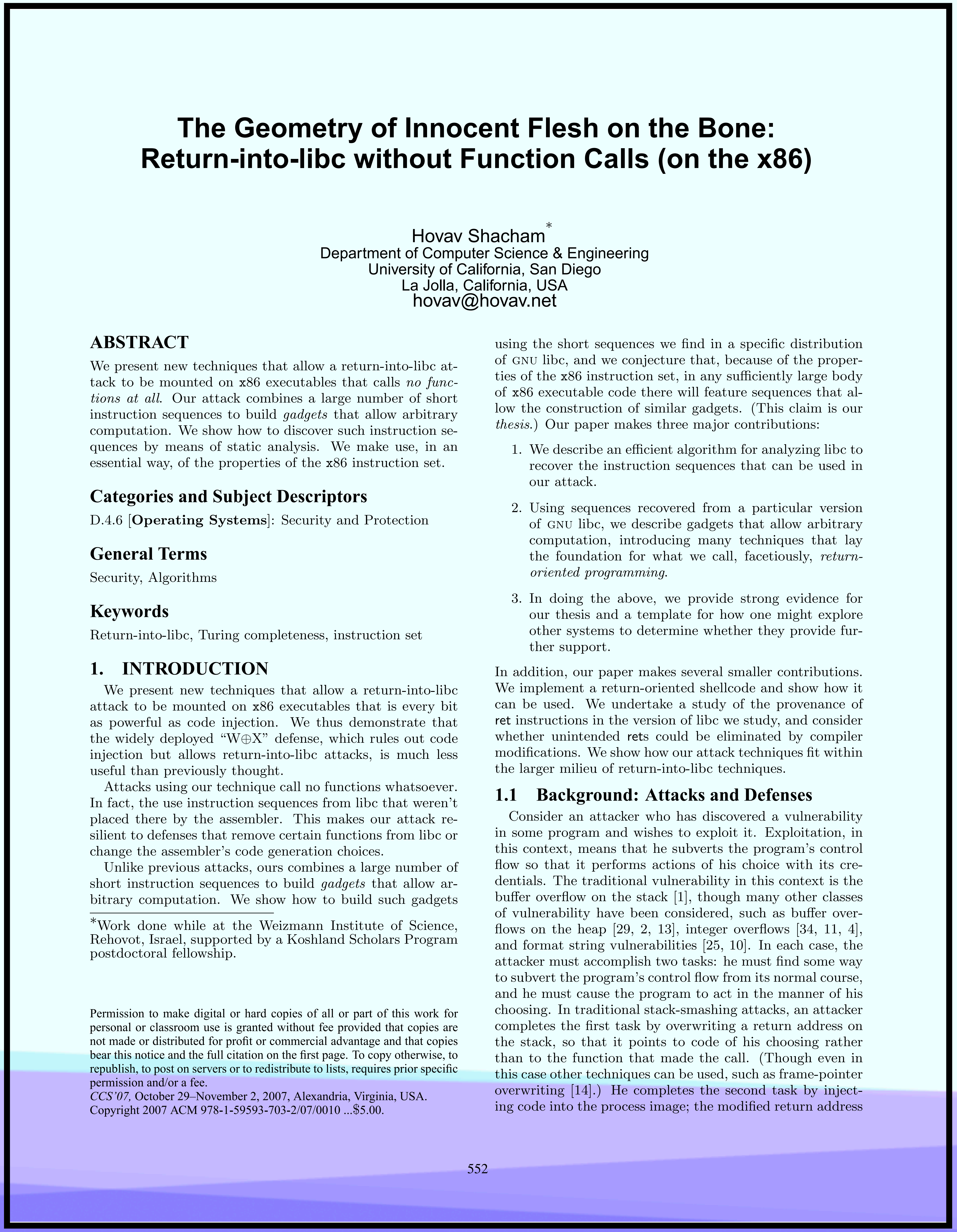
The Geometry of Innocent Flesh on the Bone : Return-into-libc without Function Calls (on the x86)

by - Hovav Shacham, UCSD | ACM CCS '07

Fitzgerald (Fiji) Marcelin, Joseph Remines, Saket Upadhyay, Wei Qi, Yu-Sheng Wang

Paper Overview



The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)

Hovav Shacham[*]
Department of Computer Science & Engineering
University of California, San Diego
La Jolla, California, USA
hovav@hovav.net

ABSTRACT

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that calls *no functions at all*. Our attack combines a large number of short instruction sequences to build *gadgets* that allow arbitrary computation. We show how to discover such instruction sequences by means of static analysis. We make use, in an essential way, of the properties of the x86 instruction set.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security, Algorithms

Keywords

Return-into-libc, Turing completeness, instruction set

1. INTRODUCTION

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that is every bit as powerful as code injection. We thus demonstrate that the widely deployed "W⊕X" defense, which rules out code injection but allows return-into-libc attacks, is much less useful than previously thought.

Attacks using our technique call no functions whatsoever. In fact, the use instruction sequences from libc that weren't placed there by the assembler. This makes our attack resilient to defenses that remove certain functions from libc or change the assembler's code generation choices.

Unlike previous attacks, ours combines a large number of short instruction sequences to build *gadgets* that allow arbitrary computation. We show how to build such gadgets

[*]Work done while at the Weizmann Institute of Science, Rehovot, Israel, supported by a Koshland Scholars Program postdoctoral fellowship.

using the short sequences we find in a specific distribution of GNU libc, and we conjecture that, because of the properties of the x86 instruction set, in any sufficiently large body of x86 executable code there will feature sequences that allow the construction of similar gadgets. (This claim is our *thesis*.) Our paper makes three major contributions:

1. We describe an efficient algorithm for analyzing libc to recover the instruction sequences that can be used in our attack.
2. Using sequences recovered from a particular version of GNU libc, we describe gadgets that allow arbitrary computation, introducing many techniques that lay the foundation for what we call, facetiously, *return-oriented programming.*
3. In doing the above, we provide strong evidence for our thesis and a template for how one might explore other systems to determine whether they provide further support.

In addition, our paper makes several smaller contributions. We implement a return-oriented shellcode and show how it can be used. We undertake a study of the provenance of ret instructions in the version of libc we study, and consider whether unintended rets could be eliminated by compiler modifications. We show how our attack techniques fit within the larger milieu of return-into-libc techniques.

1.1 Background: Attacks and Defenses

Consider an attacker who has discovered a vulnerability in some program and wishes to exploit it. Exploitation, in this context, means that he subverts the program's control flow so that it performs actions of his choice with its credentials. The traditional vulnerability in this context is the buffer overflow on the stack [1], though many other classes of vulnerability have been considered, such as buffer overflows on the heap [29, 2, 13], integer overflows [34, 11, 4], and format string vulnerabilities [25, 10]. In each case, the attacker must accomplish two tasks: he must find some way to subvert the program's control flow from its normal course, and he must cause the program to act in the manner of his choosing. In traditional stack-smashing attacks, an attacker completes the first task by overwriting a return address on the stack, so that it points to code of his choosing rather than to the function that made the call. (Though even in this case other techniques can be used, such as frame-pointer overwriting [14].) He completes the second task by injecting code into the process image; the modified return address