Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage

Stephen Checkoway
UC San Diego

Ariel J. Feldman
Princeton

Brian Kantor
UC San Diego

J. Alex Halderman
U Michigan

Edward W. Felten
Princeton

Hovav Shacham
UC San Diego

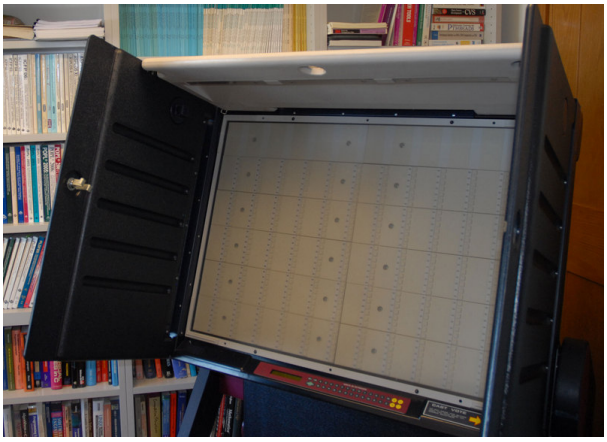
Abstract

A secure voting machine design must withstand new attacks devised throughout its multi-decade service lifetime. In this paper, we give a case study of the long-term security of a voting machine, the Sequoia AVC Advantage, whose design dates back to the early 80s. The AVC Advantage was designed with promising security features: its software is stored entirely in read-only memory and the hardware refuses to execute instructions fetched from RAM. Nevertheless, we demonstrate that an attacker can induce the AVC Advantage to misbehave in arbitrary ways — including changing the outcome of an election — by means of a memory cartridge containing a specially-formatted payload. Our attack makes essential use of a recently-invented exploitation technique called *return-oriented programming*, adapted here to the Z80 processor. In return-oriented programming, short snippets of benign code already present in the system are combined to yield malicious behavior. Our results demonstrate the relevance of recent ideas from systems security to voting machine research, and vice versa. We had no access either to source code or documentation beyond that available on Sequoia's web site. We have created a complete vote-stealing demonstration exploit and verified that it works correctly on the actual hardware.

1 Introduction

A secure voting machine design must withstand not only the attacks known when it is created but also those invented through the design's service lifetime. Because the development, certification, and procurement cycle for voting machines is unusually slow, the service lifetime can be twenty or thirty years. It is unrealistic to hope that any design, however good, will remain secure for so long.[1]

In this paper, we give a case study of the long-term security of a voting machine, the Sequoia AVC Advantage. The hardware design of the AVC Advantage dates back to the early 80s; recent variants, whose hardware differs mainly in featuring a daughterboard enabling audio voting for the blind [3], are still used in New Jersey, Louisiana, and elsewhere. We study the 5.00D version



The AVC Advantage voting machine we studied.

(which does not include the daughterboard) in machines decommissioned by Buncombe County, North Carolina, and purchased by Andrew Appel through a government auction site [2].

The AVC Advantage appears, in some respects, to offer better security features than many of the other direct-recording electronic (DRE) voting machines that have been studied in recent years. The hardware and software were custom-designed and are specialized for use in a DRE. The entire machine firmware (for version 5.00D) fits on three 64 kB EPROMs. The interface to voters lacks the touchscreen and memory card reader common in more recent designs. The software appears to contain fewer memory errors, such as buffer overflows, than some competing systems. Most interestingly, the AVC Advantage motherboard contains circuitry disallowing instruction fetches from RAM, making the AVC Advantage a true Harvard-architecture machine.[2]

Nevertheless, we demonstrate that the AVC Advantage can be induced to undertake arbitrary, attacker-chosen behavior by means of a memory cartridge containing a specially-formatted payload. An attacker who has access to the machine the night before an election can use our techniques to affect the outcome of an election by replacing the election program with another whose visible behavior is nearly indistinguishable from the legitimate program but that adds, removes, or changes votes as the attacker wishes. Unlike those attacks described

in the (contemporaneous, independent) study by Appel et al. [3, 4] that allow arbitrary computation to be induced, our attack does not require replacing the system ROMs or processor and does not rely on the presence of the daughterboard added in later revisions.

Our attack makes essential use of *return-oriented programming* [24, 8], an exploitation technique that allows an attacker who controls the stack to combine short instruction sequences already present in the system ROM into a Turing-complete set of combinators (called "gadgets"), from which he can synthesize any desired behavior. (Our exploit gains control of the stack by means of a buffer overflow in the AVC Advantage's processing of a type of auxiliary cartridge; see Section 5.) Defenses that prevent code injection, such as the AVC Advantage's instruction-fetch hardware, are ineffective against return-oriented programming, since it allows an attacker to induce malicious behavior using only preëxisting, benign code. Return-oriented programming was introduced by Shacham at CCS 2007 [24], a full two decades after the AVC Advantage was designed. Originally believed to apply only to the x86, return-oriented programming was generalized to the SPARC, a RISC architecture, by Buchanan et al. [8]. In Section 4 we show that return-oriented programming is feasible on the Z80 as well, which may be of independent interest. In addition, we show that it is possible starting with a corpus of code an order of magnitude smaller than previous work.

Using return-oriented programming, we have developed a full demonstration exploit for the AVC Advantage, by which an attacker can divert any desired fraction of votes from one candidate to another. We have tested that this exploit works on the actual hardware; but in developing our exploit we used a simulator for the machine. See Sections 5 and 6 for more on the exploit and Section 2 for more on the simulator.

Our results demonstrate the relevance of recent ideas from systems security to voting machine research, and vice versa. Our attack on the AVC Advantage would have been impossible without return-oriented programming. Conversely, the AVC Advantage provides an ideal test case for return-oriented programming. In contrast to Linux, Windows, and other desktop operating systems, in which the classification of a process' memory into executable and nonexecutable regions can be changed through system calls, the AVC Advantage is a true Harvard architecture: ROM is executable, RAM is nonexecutable.[3] The corpus of benign instruction on which we draw is just 16 kB, an order of magnitude smaller than in previous attacks.

In designing our attack, we had access neither to source code nor to usage documentation; through reverse engineering of the hardware and software, we have reconstructed the functioning of the device. This is in contrast to the Appel et al. report, whose authors did have this access, as well as to most of the previous studies of voting machines (discussed in Section 1.1 below). We had access to an AVC Advantage legitimately purchased from a government surplus site by Andrew Appel [2] and a memory cartridge similarly obtained by Daniel Lopresti. Since voting machines are frequently left unattended (as Ed Felten has documented, e.g., at [12]), we believe that ours represents a realistic attack scenario. We hope that our results go some way towards answering the objection, frequently raised by vendors, that voting security researchers enjoy unrealistic access to the systems they study.[4]

1.1 Related work

Much of the prior research on voting machine security has relied on access to source code. The first such work by Kohno et al. [18] analyzed the Diebold[5] AccuVote-TS voting machine and found numerous problems. The authors had no access to the voting machine itself but the source code had appeared on the Internet. Many of the issues identified were independently confirmed with real voting machines [9, 21, 22].

Follow up work by Hursti examined the AccuVote-TS6 and AccuVote-TSx voting machines using "source code excerpts" and by testing the actual machines. Backdoors were found that allowed the system to be extensively modified [17]. Hursti's attacks were confirmed and additional security flaws were discovered by Wagner et al. [27].

In 2006, building on the previous work, Feldman et al. examined an AccuVote-TS they obtained. The authors did not have the source code, but they note that "the behavior of [the] machine conformed almost exactly to the behavior specified by the source code to BallotStation version 4.3.1" which was examined by Kohno et al. In addition to confirming some of the security flaws found in the previous works, they demonstrated vote stealing software and a voting machine virus that spreads via the memory cards used to load the ballot definition files and collect election results [11].

In 2007, California Secretary of State Debra Bowen decertified and then conditionally recertified the direct recording electronic voting machines used in California as part of a top-to-bottom review. As part of the recertification, voting machine vendors were required to make available to independent reviewers documentation, source code, and several voting machines. In all cases, significant problems were reported with the procedures, code, and hardware reviewed [6].

Also in 2007, Ohio Secretary of State Jennifer Brunner ordered project EVEREST — Evaluation and Validation of Election Related Equipment, Standards and Testing — as a comprehensive review of Ohio's electronic voting

machines. Similar to California's top-to-bottom review, the reviewers had access to voting machines and source code. Again, critical security flaws were discovered [7].

2 The road to exploitation

In 1997, Buncombe County, North Carolina, purchased a number of AVC Advantage electronic voting machines for $5200 each. In January 2007, they retired these machines and auctioned them off through a government-surplus web site. Andrew Appel purchased one lot of five machines for $82 in total [2].

Reverse-engineering the voting machine. Two members of our team immediately began reverse engineering the hardware and software. The machine we examined is an AVC Advantage revision D. It contains ten circuit boards, including the motherboard shown in Figure 1, with an eleventh inside the removable memory cartridge — see below. Each is an ordinary two-sided epoxy-glass type. Since these are somewhat translucent, with the use of a bright light, magnifying glass, low-voltage continuity tester, and data sheets for the components, we were able to trace and reconstruct the circuit schematic diagram, and from that deduce how the unit worked. We filled in remaining details by partially disassembling the machine's software using IDA Pro.

After approximately six man-weeks of labor, we produced a functional specification [14] describing the operation of the hardware from the perspective of software running on the machine. We documented 47 I/O functions that the processor can execute to control hardware functions, such as mapping areas of ROM into the address space, interfacing with the voter panel and operator controls, and reading or writing to the memory cartridge.

Reverse-engineering the results cartridge. The AVC results cartridge is a plastic box about the dimensions of a paperback book with a common "ribbon-style" connector on one end that mates to the voting machine. Inside, there is an ordinary circuit board containing static RAM chips — backed by two type AA batteries — and common TTL 74-series integrated circuits. There is no microcontroller; instead all control signals come directly from the voting machine. Much of the internal circuitry appears to have been designed to withstand hot-plugging and to prevent accidental glitching of the memory contents.

There is an additional 8 bit of nonmemory data that can be read from the unit corresponding to the type and revision of the memory cartridge. This data is set by etch jumpers on the circuit board. We were able to change the type and revision of the cartridge by cutting the associated trace on the circuit card and wiring alternate jumpers.

The contents of memory can be read or written by powering the device and toggling the appropriate input signals. We constructed a simple microcontroller circuit to interface with the cartridge to perform reads and writes. The microcontroller simply controls the appropriate signals on the cartridge connector to perform the operation indicated by a controlling program communicating with the microcontroller via a serial port. No access to the inside circuitry was necessary.

By disassembling the software and looking at the contents of a valid results cartridge, we were able to understand the format of the file system used on the memory cartridges (and also the internal file system of the 128 kB SRAM described below) and many of the files used by the voting machine.

Crafting the exploit. Joshua Herbach used the hardware functional specifications to develop a simulator for the machine [15], which another member of our team subsequently improved.[6] Our simulator now provides cycle-accurate emulation of the Z80, and it executes the AVC election software without any apparent flaws.

We developed our exploit almost entirely in the simulator, only returning to the actual voting machine hardware at the end to validate it. Remarkably, the exploit worked the first time we tried it on the real hardware.

Total cost. Starting with no source code or schematics, we reverse engineered the AVC Advantage and developed a working vote-stealing attack with less than 16 man-months of labor. We estimate the cost of duplicating our effort to be about $100,000, on the private market.

3 Description of the AVC Advantage

In this section, we give a description of the hardware and software that makes up the AVC Advantage in some detail. Readers not interested in such low-level details are encouraged to skip ahead to Section 4, referring back to this section for details as needed.

3.1 Software

The core of the version-5.00D AVC Advantage is a Z80 CPU and three 64 kB erasable, programmable ROMs (EPROMs) which contain both code and data for the Advantage. Each EPROM is divided into four 16 kB segments: BIOS, System Toolkit, Toolkit 2, Toolkit 3, Election Program, Election Toolkit, Reports Program, Consolidation Program, Ballot Verify Program, Define Ballot Program, Maintenance Utilities, and Setup Diagnostics; see Figure 2.

When the Advantage is powered on, execution begins in the BIOS at address `0x0000`. The BIOS contains a mixture of hand-coded assembly and compiler generated code for interrupt handling, remapping parts of the address space (see Section 3.2), function call prologues and epilogues, thunks for calling code in other segments, and code for interacting with the peripherals.

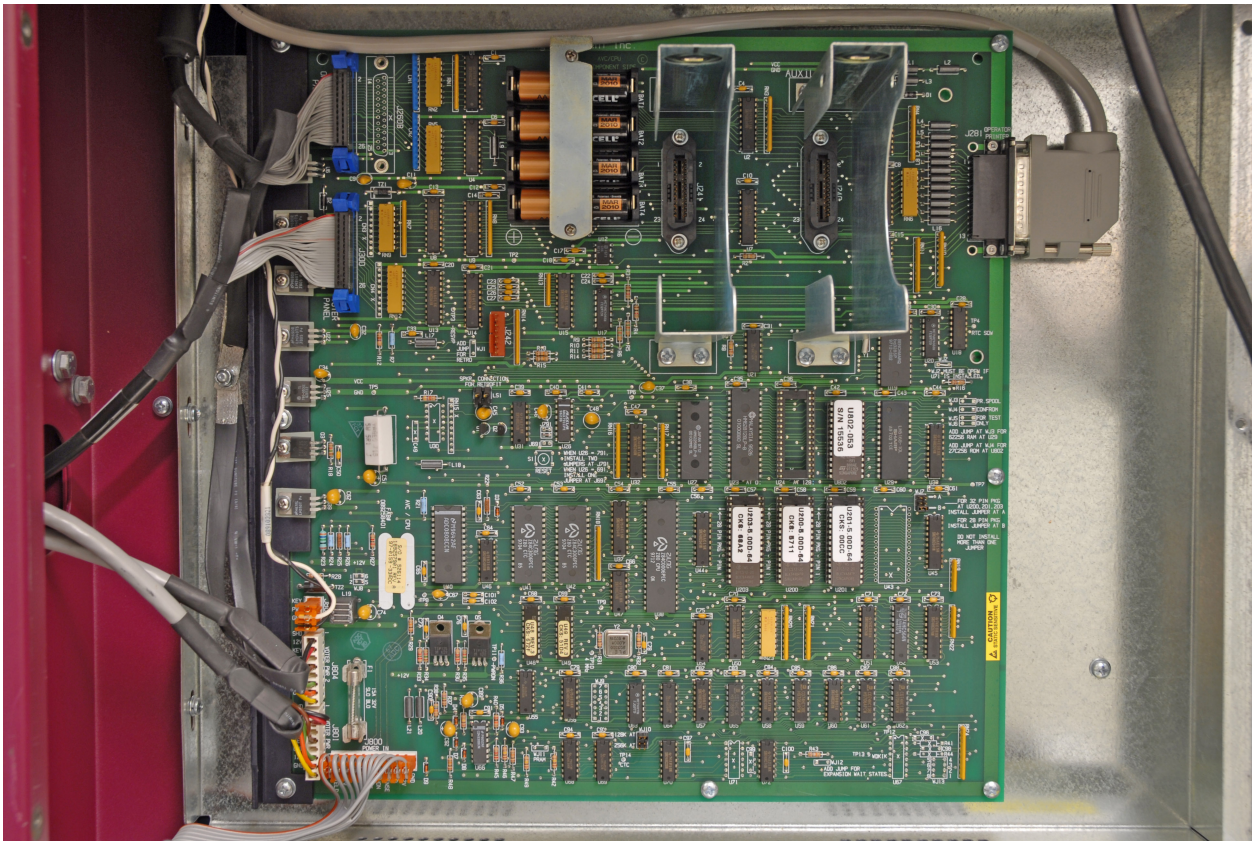


Figure 1: We reverse engineered the AVC Advantage hardware. The motherboard, shown here, is composed mostly of discrete logic and measures 14 in × 14 in. Election software is stored in removable ROM chips (white labels). The results and auxiliary memory cartridges are plugged directly into the motherboard (upper right).

Apart from the BIOS, each EPROM segment contains a 16 B header followed by a mixture of (mostly) compiler-generated code and data. The segments with "Toolkit" in their name[7] in addition to the Reports Program consist of the header followed immediately by a sequence of `jp addr` instructions, each of which jumps to a global function in the segment. For the entries in this sequence corresponding to global functions, there is a corresponding thunk in the BIOS which causes the segment to be mapped into the address space before transferring control to the function. Functions in one segment can call global functions in another segment by way of the thunks.

Each of the remaining segments is a self-contained program with just a single entry point immediately after the segment header. When a program is run, much of the state of the previous program — including the stack and the heap — is reset. In particular, any data written to the stack during one program's execution are lost during a second program's execution.

A typical sequence of events for an election would include the following. The machine is powered on and begins executing in the BIOS. The BIOS performs some initialization and tests before transitioning to a menu in Maintenance Utilities awaiting operator input. The operator selects the *Setup Diagnostics* choice and the corresponding Setup Diagnostics program is run. This performs various software and hardware tests before transitioning to the Define Ballot Program. This program checks the memory cartridge inserted into the machine and upon finding a ballot definition transitions to the Ballot Verify Program. The Ballot Verify Program checks that the format of the ballot is correct and ensures that the files which hold the vote counts are empty. After this, it illuminates the races and candidates so that the technician can verify that they are correct. Assuming everything is correct, control transfers to the Election Program for the pre-election logic and accuracy testing. The voting machine is powered off at this point and shipped to the polling places. After it has been powered back on, control again passes to the Election Program, this time for the official election.

The ZiLOG Z80 CPU is an 8 bit accumulator machine. All 8 bit arithmetic and logical operations use the accumulator register **a** as a source register and the destination register. Apart from the accumulator register, there are six general purpose 8 bit registers **b**, **c**, **d**, **e**, **h**, and **l** which can be paired to form three 16 bit registers **bc**, **de**, and **hl**. These registers along with an 8 bit flags regis-

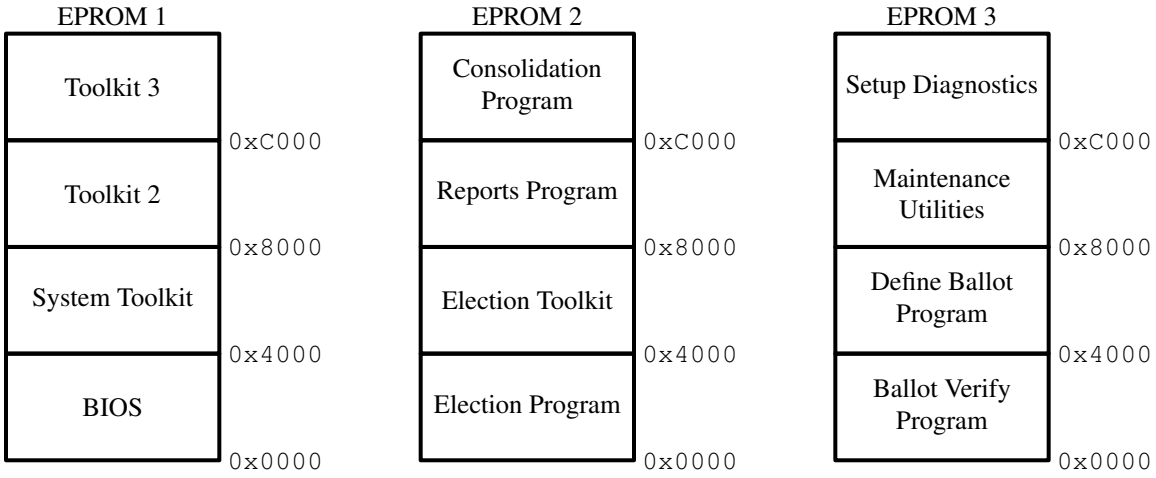
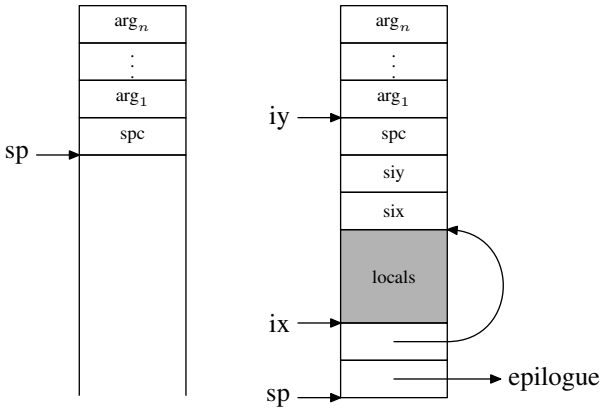


Figure 2: EPROM segment layout.

ter **f** and 16 bit stack pointer **sp** and program counter **pc** registers are compatible with the Intel 8080. In addition, there are two 16 bit index registers **ix** and **iy**, an interrupt vector register **i**, a DRAM refresh counter register **r**, and four *shadow* registers **af′**, **bc′**, **de′**, and **hl′** which can be swapped with the corresponding nonshadow registers. The Advantage uses the shadow registers for the interrupt handler which obviates the need to save and restore state in the interrupt handler. See [28] for more details.

Due to the limited ROM space for code and data, compiler-generated functions which take arguments or have local variables use additional functions to implement the function prologue and epilogue. The prologue pushes the **iy** and **ix** registers and decrements the stack pointer to reserve room for local variables. It then sets **iy** to point to the first argument and **ix**−80**h** to point to the bottom of the local stack space. Finally, it pushes the stack-address of the two saved index registers and the address of the epilogue function before jumping back to the function that called the prologue. See Figure 3. The epilogue function pops the saved pointer to the index registers and loads **sp** with it. Then **ix** and **iy** are popped and the epilogue returns to the original saved **pc**. It is the caller's responsibility to pop the arguments off the stack once the callee has returned.



(a) The state of the stack immediately after calling the function.

(b) The state of the stack after returning from the prologue function.

Figure 3: The state of the stack after calling a function with *n* arguments.

3.2 Address space layout

The AVC Advantage has a 16 bit flat address space divided into four distinct regions. The bottom 16 kB is mapped to the BIOS. The 16 kB–32 kB range can be mapped to one of the 12 16 kB aligned segments on the three program EPROMs. This mapping is controlled by the software using the Z80's `out` instruction. The 32 kB–63 kB range addresses the bottom 31 kB of a 32 kB, battery-backed SRAM. Finally, the top 1 kB of the address space can be mapped to either the top 1 kB of the 32 kB SRAM or it can be mapped to any 1 kB aligned region of a 128 kB, battery-backed SRAM. This mapping can be changed by the software using the Z80's `out` instruction. For more detail, see [14].

The AVC Advantage's stack starts at address `0x8FFE` and grows down toward smaller addresses. The heap occupies a region of memory starting from an address specified by the currently active program to `0xEBFF`. Scattered throughout the rest of 32 kB main memory, there are various global variables and space for the string table of the active program. In addition, starting at `0x934E` and growing down, there is space for a module call stack which allows modules to make calls to functions in other modules, such as `printf` or `strcpy`. See Figure 4.

As the lower 32 kB of the address space corresponds to EPROMs, data cannot be written to those addresses and attempts to do so are silently ignored by the hard-

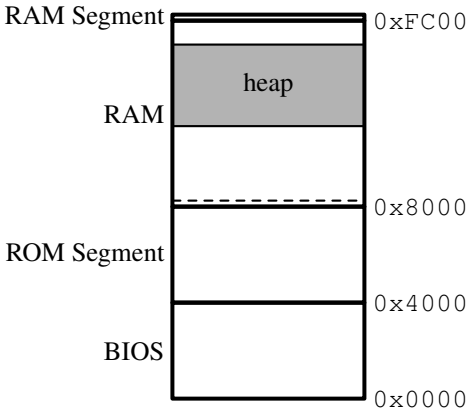


Figure 4: Address space layout of the AVC Advantage. The dashed line represents the start of the stack at `0x8FFE`. The ROM and RAM Segments are the portions of the address space mappable to the 16 kB aligned segments of the EPROM and 1 kB aligned segments of the 128 kB SRAM, respectively.

ware.[8] Similarly, as the upper 32 kB of the address space is for writable memory, not program code, any attempt to fetch an instruction from those addresses raises a non-maskable interrupt (NMI). The NMI causes the processor to load a known constant into the **`pc`** register and execution resumes in the BIOS where the processor will be halted after displaying an error message on the operator LCD. This design makes the AVC Advantage a Harvard-architecture computer.

4 Return-oriented programming

Since the AVC Advantage is a Harvard architecture computer, traditional code injection attacks cannot succeed because any attempt to read an instruction from data memory causes an NMI which will halt the machine. In practice, given a large enough corpus of code, this is not a barrier to executing arbitrary code using *return-oriented programming* — an extension of *return-to-libc* attacks where the attacker supplies a malicious stack containing pointers to short instruction sequences ending with a **`ret`** [24, 8].

The Z80 instruction set is very dense. Every byte is either a valid opcode or is a prefix byte. As there are no invalid or privileged instructions, instruction decoding of any sequence of data always succeeds. This density facilitates return-oriented programming since we can exploit unintended instruction sequences to build *gadgets* — a sequence of pointers to instruction sequences ending with a **`ret`**. For a concrete example, the BIOS contains the code fragment **`ld bc,`**`2`**`; ret`** — a potentially useful instruction sequence in its own right — which is `01 02 00 c9` in hex where the first three bytes are the load and the last is the return. If we set the program counter one byte into the load instruction, then we get the instruction sequence `02 00 c9` corresponding to the three instructions **`ld (bc),a; nop; ret`** which stores the value of the accumulator into memory at the address pointed to by the register **`bc`**.

Shacham [24] and later Buchanan et al. [8] had code corpora on the order of a megabyte from which to construct gadgets. In contrast, Francillon and Castelluccia [13] had only 1978 B of code with which to craft gadgets; however, they did not construct a Turing-complete set of gadgets. This prompts the question: What is the minimal amount of code required to construct a Turing-complete set of gadgets? By constructing a Turing-complete set of gadgets using only the AVC Advantage's BIOS — which consists of 16 kB of code and data — we make progress toward answering that question.

Following Shacham, we wrote a small program to find sequences of instructions ending in **`ret`**. We ran this program on the AVC Advantage's BIOS. We then manually devised a Turing-complete set of gadgets from the instruction sequences found by our program, including gadgets to control the peripherals like the LCDs and memory cartridges. We build a collection of gadgets that implement a 16 bit memory-to-memory pseudo-assembly language. See Table 1 for a description of the pseudo-assembly language and Appendix A for the implementation of many of the gadgets and a precise explanation of the notation that will be used in the remainder of the paper.

(We stress that demonstrating return-oriented programming on the Z80 is a major contribution of this paper and of independent interest; we have moved the details to an appendix to improve the paper's flow.)

Some of the gadgets in Table 1 are straightforward to construct; others require more finesse due to tricky interactions among the registers used in the instruction sequences. For ease of implementation, no state is presumed to be preserved between gadgets. That is, all arguments are loaded from memory into registers, operated upon, and then stored back into memory.[9] In this way, each gadget can be reasoned about independently. The operands to the gadgets are either global variables — declared with the `.var` directive — or immediate values; labels are resolved to offsets and thus are immediate values.

Some of the instruction sequences described in Appendix A contain `NUL` bytes which make them unsuitable for use in stack smashing attacks using a string copy. An early implementation of the gadgets took great pains to avoid all zero bytes. However, using the multi-stage exploit described below, avoiding zero bytes was unnecessary except for in the first stage of the exploit which did not use the gadgets presented in this section. As such, the simpler form of the gadgets is presented.

It has become traditional in papers on return-oriented programming to show a sorting algorithm implemented

Table 1: The return-oriented pseudo-assembly language for the AVC Advantage consists of seven directives and 39 mnemonics. An uppercase letter denotes a variable as defined by the `.var` directive and n denotes a 16 bit literal.
† Register **bc** is set to C and the least significant byte of A is used for the accumulator.
‡ The AVC Advantage has a watchdog timer that raises a non-maskable interrupt if it is not reset often enough. See [14].

Mnemonic	Description
`.ascii "str"`	Inserts the bytes for `str`
`.asciiz "str"`	`.ascii "str"; .byte 0`
`.byte b,...`	Insert a byte for each argument
`.data n`	Inserts n `NUL` bytes
`.var A,n`	Define a new 16 bit variable A at location n
`.word n,...`	Insert a word for each argument
`label:`	Define a new label
`add A,B,C`	$A \leftarrow B+C$
`addi A,B,n`	$A \leftarrow B+n$
`and A,B,C`	$A \leftarrow B \,\&\, C$
`b label`	Branch to `label`
`btr A,label`	Branch to `label` if A is true
`bfa A,label`	Branch to `label` if A is false
`call SP,label`	Push address of the next gadget to stack at SP, jump to `label`
`cpl A,B`	$A \leftarrow \sim B$
`dec A`	$A \leftarrow A-1$
`di`	Disable interrupts
`ei`	Enable interrupts
`halt`	Halt the machine
`in A,C`	**`in a,(c)`**†
`inc A`	$A \leftarrow A+1$
`jr A`	Jump to address A

Mnemonic	Description
`la A,B`	Set A to the address of B
`la A,label`	Set A to the address at `label`
`ld A,n(B)`	$A \leftarrow (B+n)$
`ldx A,B,C`	$A \leftarrow (B+C)$
`li A,n`	$A \leftarrow n$
`mov A,B`	$A \leftarrow B$
`mul A,B,C`	$A \leftarrow B \times C$
`neg A,B`	$A \leftarrow -B$
`nop`	Do nothing
`or A,B,C`	$A \leftarrow B \mid C$
`out C,A`	**`out (c),a`**†
`pet`	Pet the watchdog timer‡
`pop SP,A`	$A \leftarrow (SP)$; $SP \leftarrow SP+2$
`push SP,A`	$SP \leftarrow SP-2$; $(SP) \leftarrow A$
`pushi SP,n`	$SP \leftarrow SP-2$; $(SP) \leftarrow n$
`ret SP`	Pop from stack at SP, jump to value
`seq A,B,C`	$A \leftarrow B = C$
`slt A,B,C`	$A \leftarrow B < C$
`slti A,B,n`	$A \leftarrow B < n$
`sne A,B,C`	$A \leftarrow B \neq C$
`srl A,B,s`	$A \leftarrow B \gg s$
`st A,n(B)`	$(B+n) \leftarrow A$
`stx A,B,C`	$(B+C) \leftarrow A$
`sub A,B,C`	$A \leftarrow B-C$

as a return-oriented program [8, 16]. In Appendix B, we give the listing for a return-oriented Quicksort. We have verified that this sorting algorithm works on the actual AVC Advantage as part of a larger program that prints a list of numbers on the printer, sorts them, and prints the sorted list.

5 A multi-stage exploit for the AVC Advantage

Even though many parts of the code we reverse engineered appear to handle data from memory cartridges safely, we have been able to find a stack buffer overflow vulnerability. In this section, we describe this vulnerability and discuss how an attacker can exploit it to overwrite the AVC Advantage's stack and reliably induce the execution of a return-oriented payload of his choice.

We stress that the buffer overflow that we have identified appears to be unrelated to the one identified by Appel et al. in their report [3, Section II.26]. Our buffer overflow occurs in cartridge processing whereas Appel et al.'s occurs in interaction with the daughterboard (which the machine we studied lacks); our overflow requires manual action, whereas Appel et al.'s is triggered on boot; our overflow is exploitable for diverting the machine's control flow, whereas Appel et al.'s appears to allow only a denial of service. We do not know whether the overflow that we found persists in the more recent AVC Advantage version that Appel et al. examined.

One of the programs not normally used in an election, but accessible from the main menu, contains a buffer overflow while reading from an auxiliary cartridge of a certain type. (As described in Section 2, we physically modified a results cartridge so that the AVC Advantage would recognize it as a cartridge of the type for which the appropriate menu item is enabled.) A maliciously crafted field in one of the files allows roughly a dozen bytes to be written at the location of the saved stack pointer. In the first stage of the exploit, the **hl** register is set and the stack pointer is modified using the `sp ← hl` instruction sequence, inducing a return-oriented jump to an attacker-controlled location in memory.

For stage two, a section of memory under attacker control needs to contain gadgets. Fortunately (for the attacker), a file of fixed size but with several dozen unused bytes is read from the memory cartridge into a buffer allocated by `malloc`. By the time of the overflow in stage

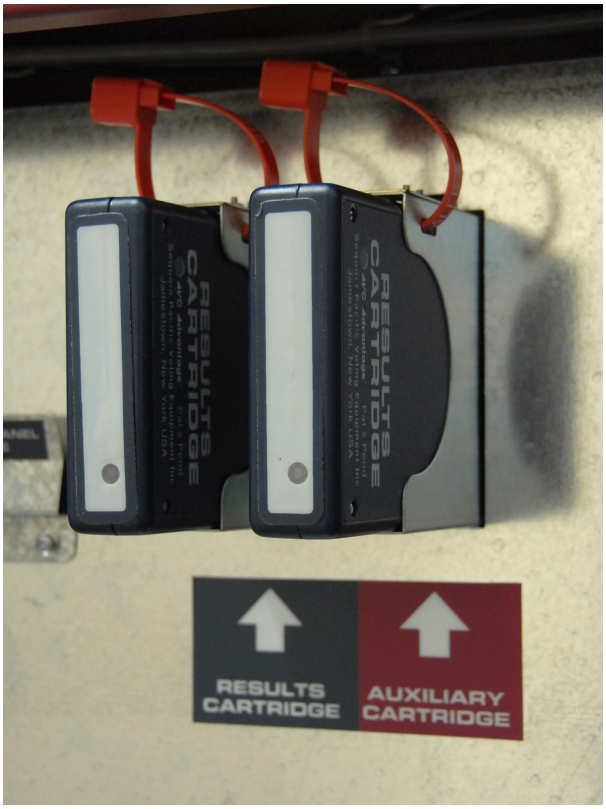


Figure 5: The machine has slots for two memory cartridges. The first cartridge stores ballots and votes. An attacker could install vote sealing code by inserting a prepared cartridge into the second slot.

one, this buffer has been deallocated but most of its contents remain in memory at a known location. This unused space can be changed to contain gadgets that make up the second stage of the exploit. The first thing that stage two does is reallocate memory for the buffer so that additional allocations will not overlap and thus write over the gadgets. At the same time, enough memory is allocated to hold the contents of an additional file from the memory cartridge. The data from this file — stage three of the exploit — is read into the allocated buffer. Control then transfers to stage three which can perform arbitrary code execution using the gadgets described in Section 4.

We have tested on an AVC Advantage that the exploit procedure described in this section works, using it both to run the sorting program described in the previous section and the vote-stealing exploit described in the next.

6 Using the exploit to steal votes

We have designed and implemented a demonstration vote-stealing exploit for the AVC Advantage, using the vulnerability described in the previous section to take over the machine's control flow. We have tested that our exploit works on the actual AVC Advantage. (Although it was designed and debugged exclusively in our simulator, the exploit worked on the real hardware on the first try.) In this section, we describe both the actions that an attacker will undertake to introduce the exploit payload to the machine and the behavior of the payload itself. We also note several ways in which the exploit could be made more resistant to detection by means of forensic investigation.

Our attacker accesses the AVC Advantage when it is left unattended the night before the election. Ed Felten has described how such access is often possible (see, e.g., [12]). At this point, the machine has been loaded with an election definition and has passed pre-LAT.[10] The attacker picks the locks for the back cabinet, the voter panel, and (later) the open/close polls switch. Appel et al. have shown that these locks are of a low-security kind that is easily bypassed [3, Section I.9]. The attacker does not need to remove any tamper-evident seals; in particular, he does not need to remove the circuit-board cover.

Having gained access to the back cabinet of the AVC Advantage, the attacker uses the normal functions to open the polls, cast a single vote, and close the polls. (The polls cannot be closed with no votes cast.[11]) Once the polls are closed, the attacker unseats the results cartridge. The cartridge cannot be removed completely because of the tamper-evident seal; however, the seal is small enough compared to the holes through which it is inserted that the cartridge can be disconnected from the machine. With the polls closed and the cartridge removed, the attacker uses the two-key reset gesture ("print-more" and "test") to gain access to the machine's post-election menu. From this menu, he can reset the machine; after the reset, the machine's main menu is accessible. (Were the results cartridge not removed, the data on it would be erased by the reset. The attacker might be able to recreate this data and rewrite it to the results cartridge, but unseating the cartridge before the reset obviates this.)

To this point, the attacker is simply following the same procedures poll workers and election officials use in running an election and resetting the AVC Advantage for the next election. His goal is to gain access to the main menu, from which he can direct the machine toward the vulnerability described in Section 5.

The system reset appears to clear the audit logs on the machine. Our demonstration vote-stealing exploit does not undo this log-clearing, though a more stealthy attack might wish to; otherwise, a post-election audit might discover that log entries are missing. (Although, as Davtyan et al. have found in their audit of the AccuVote AV-OS system [10], discrepancies in logs are not uncommon and may not be perceived as signs of an attack.) Even if the attack is detected, the original voter intent will not be recoverable. The attacker can use the post-election menu to dump the contents of the logs either to a trans-

fer cartridge or to the printer and cause his exploit payload to restore them once the system is compromised. In addition, since a vote was cast, the protective counter has been incremented; however, the protective counter is subject to software manipulation and could easily be rolled back if the attacker desires. Traces of the phantom vote might also remain in the machine or operator logs; if so, a stealthy exploit would have to remove these traces.

The attacker now reinserts the results cartridge and a cartridge of the appropriate type into the auxiliary port and navigates the menus to trigger the vulnerability described in Section 5. Using a three-stage exploit as described in Section 5, he takes control of the AVC Advantage and can execute arbitrary (return-oriented) code.

Note that hardware miniaturization since the design of the AVC Advantage makes possible the creation of cartridges much smaller than legitimate cartridges with orders of magnitude more storage. (Different parts of memory could be paged in using a "secret knock" protocol.) A smaller cartridge may allow the attacker to bypass tamper-evident loops placed on the auxiliary port guide rails that would prevent the insertion of a legitimate cartridge (although we are not aware of a jurisdiction that attempts to limit access to the auxiliary port in this way); it may also allow him to leave an auxiliary cartridge in place during voting while avoiding detection, which would be useful for exploit payloads larger than can fit in main memory and unused portions of the results cartridge. (As noted below, our exploit payload easily fits in main memory.)

The exploit first restores those parts of the machine's state necessary to allow the election to begin again. It copies the results cartridge's post-LAT voting files (which are in their empty state) over the results cartridge's election files so that the single ballot that was cast in order to close the polls is erased. It then copies (most of) the contents of the results cartridge into the internal memory. At this point, a message is displayed on the operator LCD instructing the attacker to remove the auxiliary cartridge and turn off the power.

In order to convincingly simulate power off, we need the power switch to be in the off position. Luckily, the AVC Advantage has a *soft* power switch, so turning the power knob just sets a flag that can be polled by the processor at interrupt time to detect power off. So long as the exploit code disables interrupts (while petting the watchdog timer to keep it from firing) it can keep the machine running; it can also detect when, later, the power switch is turned to the on position. (By contrast, were the machine actually to power down, the stack would be reset on a subsequent power up and the attacker would lose control.) The AVC Advantage features a large 110 V battery designed for 16 hours of operation that we believe will allow it to remain overnight in this state [23]. Of all the steps in our exploit, this is the one that most intimately relies on the details of the AVC Advantage's hardware implementation. We emphasize that we have tested on the actual machine that our exploit code is able to survive a power-down/power-up cycle in this way on battery power alone.

When the exploit code detects that the power switch has been turned to the off position, it simulates power down. It turns off the LEDs in the voter panel, clears the LCD displays, and turns off any status LEDs. In testing on an AVC Advantage, we have been able to disable (via return-oriented code) all indicators of power except the LCD backlight on the operator panel. This is the most visible sign of our attack; we are currently studying how the backlight might be disabled.

The attacker now closes and locks the operator and voter panels, removes the auxiliary cartridge, and leaves. The next morning, poll workers open the machine and use the power switch to turn it on. The exploit code detects the change and simulates the machine's power-up behavior, followed by the official election mode messages.

The exploit must now simulate the machine's normal behavior when poll workers open and close the polls and when voters cast votes. While it would be possible to reimplement this behavior entirely using return-oriented code, the design of the AVC Advantage's voting program makes it possible for us to reuse large portions of the legitimate code, making the exploit smaller, simpler, and more robust. This would be more difficult to do if the exploit modified votes as they were cast, but we have instead chosen to wait until polls are closed and only then change the cast votes retroactively. The absence of a paper audit trail means that the vote modification will not be detected. Other possible designs for vote-stealing software are described by Appel et al. [3, Section I.5–6].

The main voting function is structured as a series of function calls that can be separated into three main groups, each called a single time in order in the normal case. The first group of functions waits for the "open/-close polls" switch to be set to open and prints the zero tape. The second group of functions handles all of the voting, including waiting for the activate button to be pressed and handling all voter input. Once the polls are closed, the third group of functions handles printing the final results tape and all post-election tasks.

Our demonstration exploit uses the high-level functions in the AVC Advantage's legitimate voting program to handle all voting until the polls are closed. Then the exploit reads the vote totals, moves half of the votes for the second candidate to the first candidate, and changes the cast vote records (CVRs) to match the vote totals. (Obviously, any fraction of the votes could be modified. Furthermore, while our exploit processes the CVR log in

order, changing every CVR cast for the disfavored candidate until the desired shift has been effected, more sophisticated strategies are possible.) The exploit now relinquishes control for good, handing control over to the legitimate AVC Advantage program to handle all post-election behavior. When the "Official Election Results Report" is printed it will reflect the results as modified by the exploit.

The AVC Advantage contains routines to check the consistency of its internal data structures. When the data is inconsistent, e.g., the vote totals do not match the CVR totals, this is noted in the Results Report. The exploit ensures that all data structures in memory and on the results cartridge that are checked by these routines are consistent whenever the routines are executed.

Even after it has relinquished control, our exploit remains in main memory until the machine is shut down. Forensic analysis of the contents of the AVC Advantage's RAM would be a nontrivial task; nevertheless, a stealthier exploit would wipe itself from memory before returning control to the legitimate program. If any portion of the exploit code is stored on a cartridge, this must be wiped as well. Because suspicious poll workers might remove the cartridge before it can be wiped, anything stored on a cartridge should be kept encrypted, and the exploit code should scrub the key from RAM if it detects that the cartridge has been removed.

Our vote-stealing demonstration exploit is just over 3.2 kB in size, including all of the code to copy the files and the memory cart. It fits entirely in RAM, as would even a substantially more sophisticated exploit: There is roughly an additional 10 kB of unused heap space that could be used. In addition, any code that is executed only while the attacker is present need not actually stay in the heap once it is finished and could be replaced with additional code for modifying the election outcome.

7 Conclusions

A secure voting machine design must withstand attacks devised throughout the machine's service lifetime. Can real designs, even ones with promising security features, provide such long-term security? In this paper, we have answered this question in the negative in the case of the Sequoia AVC Advantage (version 5.00D). We have demonstrated that an attacker can exploit vulnerabilities in the AVC Advantage software to install vote-stealing malware by using a maliciously-formatted memory cartridge, without replacing the system ROMs. Starting with no source code, schematics, or nonpublic documentation, we reverse engineered the AVC Advantage and developed a working vote-stealing attack with less than 16 man-months of labor. Our exploit relies in a fundamental way on return-oriented programming, a technique introduced some two decades after the AVC Advantage was designed. In mounting the attack, we have extended return-oriented programming to the Z80 processor.

Acknowledgments

We thank Andrew Appel for allowing us to use his AVC Advantage machines, and for helpful discussions about election procedures. We thank Daniel Lopresti for lending us an AVC Advantage results cartridge. We thank Joshua Herbach for developing the initial version of the AVC Advantage simulator [15]. We thank Eric Rescorla and Stefan Savage for helpful discussions.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Andrew W. Appel. How I bought used voting machines on the Internet, February 2007. `http://www.cs.princeton.edu/~appel/avc/`.

[3] Andrew W. Appel, Maia Ginsburg, Harri Hursti, Brian W. Kernighan, Christopher D. Richards, and Gang Tan. Insecurities and inaccuracies of the Sequoia AVC Advantage 9.00H DRE voting machine, October 2008. Online: `http://citp.princeton.edu/voting/advantage/advantage-insecurities-redacted.pdf`.

[4] Andrew W. Appel, Maia Ginsburg, Harri Hursti, Brian W. Kernighan, Christopher D. Richards, Gang Tan, and Penny Venetis. The New Jersey voting-machine lawsuit and the AVC Advantage DRE voting machine. In David Jefferson, Joseph Lorenzo Hall, and Tal Moran, editors, *Proceedings of EVT/WOTE 2009*. USENIX/ACCURATE/IAVoSS, August 2009.

[5] Yonatan Aumann, Yan Zong Ding, and Michael Rabin. Everlasting security in the bounded storage model. *IEEE Trans. Info. Theory*, 48(6):1668–80, June 2002.

[6] California Secretary of State Debra Bowen. "Top-to-Bottom" Review of voting machines certified for use in California. Technical report, California Secretary of State, 2007. `http://sos.ca.gov/elections/elections.vsr.htm`.

[7] Ohio Secretary of State Jennifer Brunner. Evaluation & Validation of Election-Related Equipment, Standards & Testing. Technical report, Ohio Secretary of State, 2007. `http://www.sos.state.oh.us/SOS/Text.aspx?page=4512`.

[8] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In Paul Syverson and Somesh Jha, editors, *Proceedings of CCS 2008*, pages 27–38. ACM Press, October 2008.

[9] Compuware Corporation. Direct recording electronic (DRE) technical security assessment report, November 2003.

[10] Seda Davtyan, Sotiris Kentros, Aggelos Kiayias, Laurent Michel, Nicolas Nicolaou, Alexander Russell, Andrew See, Narasimha Shashidhar, and Alexander A. Shvartsman. Pre-election testing and post-election audit of optical scan voting terminal memory cards. In David Dill and Tadayoshi Kohno, editors, *Proceedings of EVT 2008*. USENIX/ACCURATE, July 2008.

[11] Ariel J. Feldman, J. Alex Halderman, and Edward W. Felten. Security analysis of the Diebold AccuVote-TS voting machine. In Ray Martinez and David Wagner, editors, *Proceedings of EVT 2007*. USENIX/ACCURATE, August 2007. `http://itpolicy.princeton.edu/voting/ts-paper.pdf`.

[12] Edward W. Felten. NJ election day: Voting machine status, June 2008. `http://www.freedom-to-tinker.com/blog/felten/nj-election-day-voting-machine-status`.

[13] Aurélien Francillon and Claude Castelluccia. Code injection attacks on Harvard-architecture devices. In Paul Syverson and Somesh Jha, editors, *Proceedings of CCS 2008*, pages 15–26. ACM Press, October 2008.

[14] J. Alex Halderman and Ariel J. Feldman. AVC Advantage: Hardware functional specifications. Technical Report TR-816-08, Department of Computer Science, Princeton University, Princeton, New Jersey, March 2008.

[15] Joshua S. Herbach. Simulating the Sequoia AVC Advantage DRE voting machine, May 2007. `http://www.cs.princeton.edu/~herbach/SimulatingAVCAdvantage.pdf`.

[16] Ralf Hund, Thorsten Holz, and Felix Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In Fabian Monrose, editor, *Proceedings of Usenix Security 2009*. USENIX, August 2009. To appear.

[17] Harri Hursti. Critical security issues with Diebold TSx, May 2006.

[18] Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach. Analysis of an electronic voting system. In David A. Wagner and Michael Waidner, editors, *Proceedings of Security and Privacy ("Oakland") 2004*, pages 27–40. IEEE Computer Society, May 2004.

[19] Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique, September 2005. `http://www.suse.de/~krahmer/no-nx.pdf`.

[20] Tal Moran and Moni Naor. Receipt-free universally-verifiable voting with everlasting privacy. In Cynthia Dwork, editor, *Proceedings of Crypto 2006*, volume 4117 of *LNCS*, pages 373–92. Springer-Verlag, September 2006.

[21] RABA Innovative Solution Cell. Trusted agent report: Diebold AccuVote-TS voting system, January 2004.

[22] Science Applications International Corporation. Risk assessment report Diebold AccuVote-TS voting system and processes, September 2003.

[23] Sequoia Voting Systems. *AVC Advantage*. `http://www.verifiedvotingfoundation.org/downloads/AVCAdvantage.pdf`.

[24] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Sabrina De Capitani di Vimercati and Paul Syverson, editors, *Proceedings of CCS 2007*, pages 552–61. ACM Press, October 2007.

[25] Sequoia Voting Systems. Response from Sequoia Voting Systems to the California Secretary of State's office on the Top-to-Bottom Review of Voting Systems, July 2007. `http://www.sequoiavote.com/press.php?ID=32`.

[26] Sequoia Voting Systems. Response from Sequoia Voting Systems to the expert report of Andrew W. Appel evaluating the security and accuracy of the Sequoia AVC Advantage DRE voting computer, October 2008. `http://www.sequoiavote.com/documents/SVS_Response_to_Appel_report_NJ.pdf`.

[27] David Wagner, David Jefferson, and Matt Bishop. Security analysis of the Diebold AccuBasic interpreter, February 2006.

[28] ZiLOG Inc., San Jose, CA, USA. *Z80 Family CPU User Manual*, 2004.

A Implementing the gadgets

This appendix describes the construction of a number of the gadgets listed in Table 1. Many of the omitted ones are quite similar to those detailed below.

A.1 A note on notation

In what follows, `typewriter` font uppercase letters — e.g., `A` — are used to represent global variables which are literal 16 bit locations in memory. The value associated with a variable `A` is written in an *italic* font *A*. Literal numbers are written with *italic* font lowercase letters — e.g., *n*. Z80 assembly is written in a `typewriter` font with mnemonics and register names written with **`bold`** weight — e.g., **`ld b,`**`FFh`. Abbreviated instruction sequence forms (see below) and the gadget pseudo-assembly language are written in `typewriter` font — e.g., `b` ← `0xFF` and `add A,B,C`, respectively. In figures, nonabbreviated instruction sequences are boxed. In Z80 assembly, hexadecimal numbers are written with a trailing `h` as is customary. Otherwise, the C notation is followed by prepending `0x`.

Each box in the following figures of the gadgets represents a two-byte *stack slot*. Each slot contains either a literal value — either fixed for a particular gadget or the address of a global variable or code offset — or the address of an instruction sequence. The literal values are written as either hexadecimal numbers or symbolically. Addresses of instruction sequences are represented as arrows pointing to either the abbreviated form of an instruction sequence or the boxed text of the sequence it-

self. Each gadget is entered by the processor executing a **ret** with the stack pointer pointing to the bottom of the gadget. The instruction sequences are executed in order from bottom to top.

A.2 Moving data around

Any set of useful gadgets needs to contain gadgets to move data between memory and registers as well as gadgets for loading registers with constant values. At a minimum, this should include gadgets for loading immediates to registers, loading from memory into registers, moving values between registers, and storing values from registers into memory.

Loading immediate values is as simple as using instruction sequences like

```
# pop hl, de
pop hl
pop de
ret
```

which loads the next two stack slots into registers **hl** and **de**. There are instruction sequences to load each individual register as well as many combinations of registers.

Loading values from memory — e.g., from a global variable — requires loading register **hl** with the address of the variable and then using one of the two sequences

```
# bc ← (hl)
ld c,(hl)
inc hl
ld b,(hl)
ret

# hl ← (hl)
ld a,(hl)
inc hl
ld h,(hl)
ld l,a
ret
```

which load the 16 bit value pointed to by **hl** into either **bc** or **hl**.

Once the operands are in registers, other instruction sequences can operate on them. After the computation is complete, the value needs to be stored back into memory. The most common way of storing values back to memory is to place the result in register **hl** and the target address in **de**. Then the sequence

```
# (de) ← hl
ex de,hl  # swap de and hl
ld (hl),e # ⋆
inc hl
ld (hl),d
ret
```

will perform the store. Notice that if we use the sequence starting at the instruction marked with ⋆ instead, we have the instruction sequence `(hl) ← de` which is occasionally useful as well. To store a single byte into both the high and low byte of a variable, the following sequence can be used.

```
# (hl) ← a; (hl+1) ← a
ld (hl),a
inc hl
ld (hl),a
ret
```

Two simple gadgets for loading an immediate value into a variable (`li A,n`) and moving the value of one variable into another (`mov A,B`) are given in Figure 6. Rather than duplicate the full text of each instruction, common sequences are given in the abbreviated form that appears in the comments above. The `li` gadget first pops the address of variable `A` into register **hl** and the immediate value into register **de**. Then the value in **de** is stored to the address pointed to by **hl**. The `mov` gadget is similar except that **de** holds the target address — the address of variable `A` — while **hl** gets the address of variable `B` and then the value *B*.

The three main 16 bit registers **bc**, **de**, and **hl** are not interchangeable as far as our set of instruction sequences are concerned. Many operations can only be performed using a single sequence and that sequence expects its operands to be in particular registers. As a result, we need a way to move data among the three registers. To that end, we have the following three useful instruction sequences.

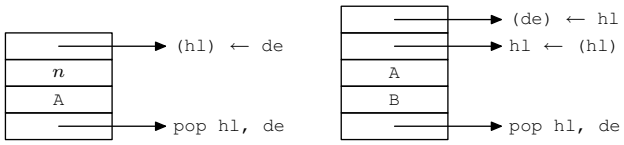
```
# bc ← hl
ld c,l
ld b,h
ret

# de ↔ hl
ex de,hl
ld bc,1
ret

# hl ← bc
ld h,b
ld b,l
ld l,c
ld c,a
ret
```

The first simply copies **hl** to **bc** without disturbing anything else. The other two destroy the contents of **bc** in the process. For `hl ← bc`, one could first use the sequence `a ← b`, `l ← a`, and `a ← c`. In this way, the contents of **bc** would be preserved. This is not necessary for the gadgets we construct.

In addition to immediate loads and moves, we implement *base plus offset* and *base plus index* loads and stores for moving data around. The base plus offset instruc-



(a) `li A,n` **(b)** `mov A,B`

Figure 6: Gadgets for loading a variable `A` with either an immediate value *n* or the value of another variable `B`.

tions `ld` (resp. `st`) take a base register and an immediate offset which is added to the base to form the source (resp. target) address. The base plus index instructions `ldx` (resp. `stx`) take a base register and an index register which are summed to form the source (resp. target) address. The implementation is a straight-forward extension of the `mov` gadget and the addition gadget described in the next subsection.

A.3 Arithmetic

We show how to perform addition. The rest of the arithmetic and logic operations are similar, apart from the multiplication, which is discussed below.

Addition can be performed by loading the addends into registers, performing the addition, and storing the result into the result variable. The following, more-generally useful instruction sequence can be used to perform the last two steps, thus saving stack space.

```
# (de) ← hl + bc
add hl,bc
ex de,hl
ld (hl),e
inc hl
ld (hl),d
ret
```

The `add` gadget is given in Figure 7 (a).

The Z80 does not contain a multiply instruction. Instead, this has to be computed in software. Because multiplication is a common operation, the BIOS contains a function which takes arguments — the multiplicands — in registers **bc** and **de** and returns with the product in register **bc**. Since register **hl** is used in the computation, it is first pushed to the stack and then popped before the function exits. The address of the instruction just after the **push hl**, is the `bc ← bc * de; pop hl` sequence. To use this sequence, we need only load **bc** and **de** with the appropriate values, taking care to load **de** first since the `de ↔ hl` sequence sets **bc** as well. The `mul A,B,C` gadget is given in Figure 7 (b).

A.4 Branching

In order to perform interesting and useful computation with gadgets, they need to be able to effect a jump or branch. Since it may not be possible to know exactly where the return-oriented-programming stack will be located in memory, it is preferable to write gadgets in a position independent manner. The way to do this is to ensure that all branching is done using relative offsets. The `pop hl` instruction sequence can be used to load **hl** with a suitable displacement *d* to the desired location. Then the sequence

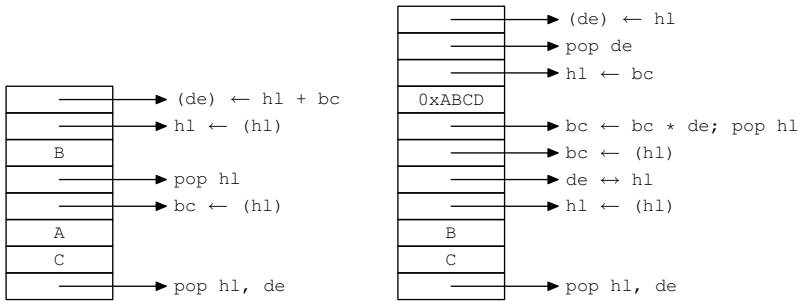
```
# sp ← sp + hl
add hl,sp
ld sp,hl
ret
```

can be used to change the stack pointer by *d* to point to another gadget. In effect, changing control to the other gadget. These two instruction sequences are exactly how the branch gadget `b label` is implemented.

Without conditional code execution, a set of gadgets can only be used to execute essentially straight-line code using Krahmer's *borrowed code chunks technique* [19] so we must have a way to do conditional branches. Following a MIPS-like ISA, we implement a *set less than* gadget `slt A,B,C` that sets *A* to `0xFFFF` if $B < C$ and `0x0000` otherwise. These values act as boolean true and false. Similarly, we implement a *set not equal* gadget `sne A,B,C` that sets *A* to `0xFFFF` if $B \neq C$ and `0x0000` otherwise.

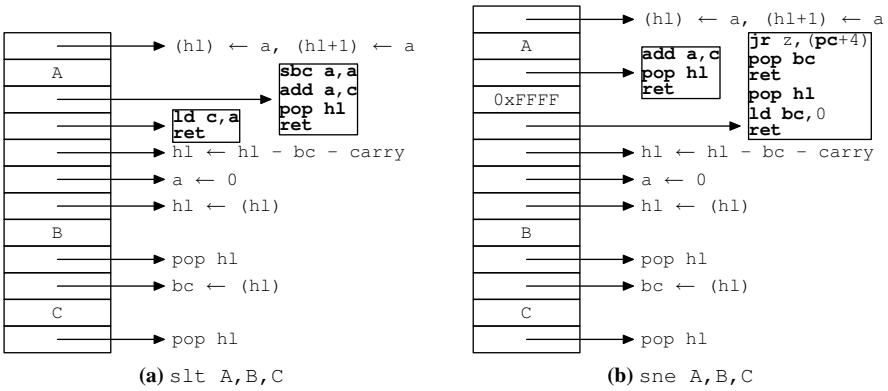
The `slt` gadget is given in Figure 8 (a). It works by first loading **bc** with the value *C* and **hl** with the value *B*. Then the accumulator **a** is cleared which has the effect of clearing the carry flag. Next, $B - C$ is computed which sets the carry flag if $B < C$. The next sequence clears **c** since **a** is zero. The penultimate sequence has the effect of setting **a** to `0xFF` if $B < C$ otherwise **a** is set to `0x00`. In addition, it loads **hl** with the variable `A`. Lastly, **a** is stored into both `(hl)` and `(hl+1)`, effectively setting *A* to either `0xFFFF` or `0x0000` depending on $B < C$ or not.

Similar to the set less than gadget, we implement a *set not equal* gadget `sne A,B,C` that sets *A* to `0xFFFF` if $B \neq C$ and `0x0000` if they are equal. The implementation is given in Figure 8 (b). It is identical to the set less than up through the subtract. After that, it uses an instruction sequence that does one of two things depending on the state of the zero flag. If $B = C$, then the subtract will set the zero flag. In this case, the **jr** `z,` (**pc**`+4`) will jump to the **pop hl** instruction causing **hl** to be loaded with `0xFFFF` and subsequently setting **bc** to `0x0000`. If $B \neq C$, then the zero flag will be cleared and **pop bc**



(a) `add A,B,C` **(b)** `mul A,B,C`

Figure 7: Arithmetic gadgets.



(a) `slt A,B,C` **(b)** `sne A,B,C`

Figure 8: Inequality tests.

will load `0xFFFF` into **`bc`**. The next instruction sequence will add **`c`** to the accumulator which was previously set to zero, thus setting **`a`** to either `0xFF` or `0x00` and load **`hl`** with the variable `A`. The final sequence sets *A* appropriately.

While not strictly necessary, a *set equal* gadget is easily constructed by adding the following sequence which sets the accumulator to its one's complement.

```
# a ← ~a
cpl
or a
ret
```

Once we have boolean values `0xFFFF` and `0x0000`, we can perform conditional branches by taking the bitwise conjunction of our boolean value and the branch offset. Due to the interactions between the registers in the available instruction sequences, performing the conjunction is tricky since the only conjunction we have available uses the accumulator and register **`c`**. Even worse, it modifies register **`bc`** in the process. Once we have computed the conjunction of a single byte, we need to place it into either register **`h`** or **`l`** depending on it being the high or low byte of the conjunction, respectively. The following three instruction sequences perform the the conjunction and the loading of **`h`** and **`l`**.

```
# a ← a & c;
# bc ← bc-1
and c
dec bc
ret

# l ← a
ld l,a
ret

# h ← a;
# l ← l + 1
ld h,a
inc l
ret
```

Since the sequence for moving the value from the accumulator to **`h`** increments **`l`**, we need to load **`h`** before we load **`l`**.

The *branch if true* gadget `btr A,label` which branches to `label` if $A = \texttt{0xFFFF}$ is given in Figure 9. If the offset from the end of the gadget to `label` is d, then let d' be the byte reversed value of d. The `btr` gadget starts by loading d' into **`bc`**. Since this value is byte-reversed, register **`c`** contains the high-order byte of the offset d. The boolean variable `A` is also loaded into **`de`**. The low-order byte of A is loaded into the accumulator

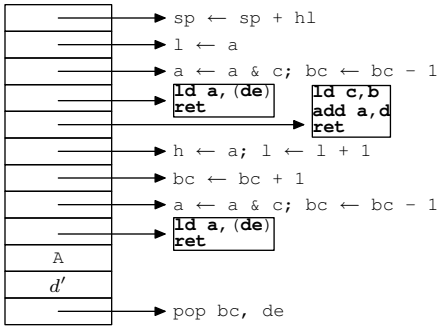


Figure 9: `btr A,label`. The immediate value d' is the byte-reversed offset d from the end of the gadget to `label`.

and the bitwise conjunction with **c** is stored into the accumulator. Since **bc** was decremented, the next instruction sequence increments it. The accumulator is then stored into **h** and **b** — the low-order byte of d — is moved into **c**. The accumulator is again loaded with the low order byte of A, the conjunction is performed and the result placed in **l**. At this point, **hl** contains the bitwise conjunction $d \,\&\, A$. If $A = \texttt{0xFFFF}$, then the next sequence will branch to the offset d. If $A = \texttt{0x0000}$, then the next sequence will do nothing.

By inserting two `a ← ~a` instruction sequences after the two `a ← (de)` sequences, a *branch if false* gadget `bfa A,label` is constructed. Once we have the `slt`, `sne`, `btr`, and `bfa` gadgets, we can perform conditional branches using numerical equality and inequality. We thus have a Turing-complete set of gadgets.

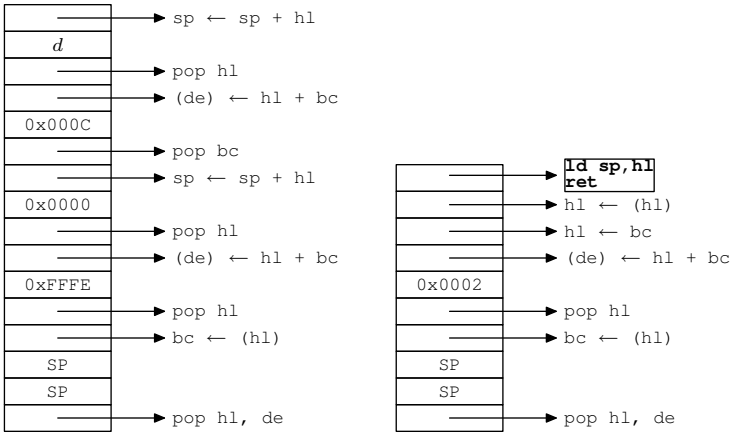
A.5 Functions

To support a more natural imperative style of programming, we implement return-oriented function calls. The return-oriented nature of our program means that the call stack is unavailable for use with return-oriented functions since our code resides on the stack. Instead, we need to designate a variable `SP` as a stack pointer for use with the stack manipulation gadgets `push`, `pop`, `call`, and `ret`.

Following the convention of the Z80, a `push SP,A` gadget first decrements SP by 2 and then stores A to (SP). A `pop SP,A` gadget first sets A to (SP) and then increments SP by 2. The `call SP,label` gadget computes the address of the following gadget and pushes that onto the stack pointed to by SP and then branches to `label`. Finally, the `ret SP` gadget pops a value off of the stack and branches to it. The `call` gadget uses the `sp ← sp + hl` instruction sequence with register **hl** set to `0x0000` to move the value of **sp** into **hl** in order to compute the address of the following gadget to push on the stack. The rest of the `call` and `ret` gadgets are straight-forward and given in Figure 10. The `push` and `pop` gadgets are similar.

B An example return-oriented program

```
.var sp,0xF000
.var array,0xF002    # saved
.var left,0xF004     # saved
.var right,0xF006    # saved
.var temp1,0xF00A    # not saved
.var temp2,0xF00C    # not saved
.var index,0xF00E    # not saved
.var i,0xF010        # not saved
.var pivot,0xF012    # not saved
qsort: # void qsort( array, left, right )
    push array
    push left
    push right
    pet
    ld left,10(sp)
    ld right,12(sp)
    slt temp1,left,right
    bfa temp1,cleanup
    ld array,8(sp)
    ldx pivot,array,right
    mov index,left
    mov i,left
loop:
    slt temp1,i,right
    bfa temp1,break
    ldx temp1,array,i
    slt temp2,pivot,temp1
    btr temp2,continue
    ldx temp2,array,index
    stx temp2,array,i
    stx temp1,array,index
    addi index,index,2
continue:
    addi i,i,2
    b loop
break:
    ldx temp1,array,index
    ldx temp2,array,right
    stx temp2,array,index
    stx temp1,array,right
    addi temp1,index,-2
    push temp1
    push left
    addi left,index,2
    push array
    call qsort
    addi sp,sp,6
    push right
    push left
    push array
    call qsort
    addi sp,sp,6
cleanup:
    pop right
    pop left
    pop array
    ret
```



(a) `call SP,label`. The immediate value *d* is the offset from the end of the gadget to `label`.

(b) `ret SP`

Figure 10: Function call and return gadgets.

As an example of performing general purpose computation, the preceding return-oriented function performs the Quicksort algorithm on its input using the gadgets from Table 1. This example "pets" the watchdog timer to keep it from firing in the middle of computation and causing a non-maskable interrupt.

Notes

[1]A related notion to long-lasting security is Moran and Naor's "everlasting privacy" requirement for cryptographic voting schemes [20], itself based on Aumann, Ding, and Rabin's "everlasting security" [5].

[2]A Harvard-architecture machine has separate data and instruction memories, in contrast to a von Neumann-architecture machine, which has a single memory for both instructions and data.

[3]Even in Francillon and Castelluccia's attack on a Mica sensor-network node [13], return-oriented programming — or, more properly, chunk borrowing à la Krahmer [19], since Turing-completeness is not necessary — is used only to fill a staging area with native code that will be installed on reboot by the bootloader.

[4]See, e.g., Sequoia's response to the Top-to-Bottom Review: "In short, the Red Team was able to, using a financial institution as an example, take away the locked front door of the bank branch, remove the security guard, remove the bank tellers, remove the panic alarm that notifies law enforcement, and have only slightly limited resources (particularly time and knowledge) to pick the lock on the bank vault" [25].

[5]Now Premier Election Solutions.

[6]Following the "Chinese wall" protocol, the simulator developers had no access to the actual hardware and relied exclusively on our published specifications.

[7]The name of a segment is contained within the segment and there is a field in the segment header which points to the name.

[8]It is possible to install a 32 kB "Program" SRAM and map the 16 kB–32 kB address range to either of the two 16 kB aligned regions of the SRAM, but no AVC Advantage used in elections has such a Program SRAM installed [26].

[9]The gadgets could be made more efficient by not writing values back to memory except as needed. This would significantly complicate hand crafting return-oriented code, but this sort of optimization is well-understood in the compiler-writing community; for example, register-allocation algorithms [1]. This sort of optimization can lead to a drastic reduction in return-oriented code size and run time.

[10]The AVC Advantage has two *Logic and Accuracy Testing* phases which are meant to test that the voting machine is in working order. The pre-LAT phase always happens prior to the election and a post-LAT phase may occur after the election, depending on policy.

[11]This is actually a configuration option. Any machines configured to allow the polls to be closed without any votes cast can skip the vote casting step. In this case, there is no need to modify the protective counter later, simplifying the exploit somewhat.