# SCROP: A Return-Oriented Programming Language

Ben Kallus[1], Charles Averill[1,2], and Zephyr Lucas[1]

[1]Department of Computer Science, Dartmouth College
[2]Department of Computer Science, UT Dallas

Abstract—Object-oriented programming (OOP) delivered us from the tyranny of understanding our programs into the Elysian realm of the FactoryBuilderBuilderFactory that we enjoy today. This feat was possible due to the development of object-oriented programming languages and their myriad beautiful abstractions, such as virtual table tables and circular inheritance. More recently, return-oriented programming (ROP) delivered us from bondage in the land of the NX bit into the wild pointer wilderness that we have wandered for the past 20 years. We present SCROP: the world's first return-oriented programming language, as a much-needed source of manna (played as an interrupt) from on high.

## I. Introduction

ROP was introduced to the academic community in Shacham's "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)" [16], a horoscope-style paper in which the reason for a reader's confusion reflects that reader's occupation. Hackers did not understand how this was any different from nergal's 6-year-old article in Phrack #58 [12], graphics researchers did not understand its connection to skeletal animation, and necromancers did not understand its connection to skeletal animation. Since then, we have been blessed with JOP [4], SROP [5], PCOP [15], DOP [10], BROP [3], STOP-DROP-AND-ROP [13], and, of course, SLOP [18]. With each of these developments, we flapped closer to Solar Designer's design, but our Icarian W^X is melting.

Much as Alan Kay is the face of OOP, despite its foundational concepts being developed by forgotten Norwegians, we aim to overshadow prior work on more useful return-oriented programming languages such as QooL by simply not citing it here and hoping that no one from David Brumley's group reads this paper.

The SCROP system consists of a compiler for a Lisp dialect that emits a stack machine assembly language, an assembler for that assembly language, and a stack machine VM in which each instruction is implemented as a ROP gadget that returns to the next instruction. That is, the output of the assembler is a ROPchain for the VM.

We summarize our contributions as follows:

1) We present SCROP, a programming language and corresponding stack machine runtime in which all programs are ROPchains, and release it under a free software license.
2) We show that programs for threaded interpreters are basically ROPchains.
3) We discover a use for the x86 ret imm16 instruction outside of callee-cleanup calling conventions.
4) We identify several factual inaccuracies in the Linux mmap(2) man page, fix one of them, and leave the rest for future work.

SCROP is implemented in 3 parts: compiler, assembler, and runtime.

The compiler is a straightforward recursive-descent/tree-walker written in Rust, loosely following Ghuloum's approach [7]. It emits a textual stack machine assembly language which is assembled by a Python script into a ROPchain for the runtime. The runtime binary is written in x86_64 GNU assembly language (GAS), using the sane syntax, and a healthy amount of C preprocessor macros. Each of these reads input from stdin and writes output to stdout, naturally leading to the following shell invocation: cat program | compiler (Rust) | assembler (Python) | runtime (GAS), which we refer to as the "natural GAS pipeline."

SCROP is available under a GPLv3 license at https://github.com/kenballus/scrop.

## II. Compiler

There's not much to say about this. It probably just behaves the way you'd guess it does.

## III. Assembler

See II.

## IV. Runtime

The SCROP runtime's job is simple: take the bytecode emitted by the assembler, and do it. However, because we want to seem like real PL researchers, we felt we had to make the runtime complicated. For this reason, we implemented it as a threaded interpreter, a style of interpreter optimized for fast instruction dispatch that has been thoroughly obsoleted by the advent of JIT compilation.

### A. VM Bytecode Format

Each SCROP VM instruction consists of a 64-bit opcode and a 64-bit immediate. Most instructions ignore the immediate, and instead take operand(s) on the VM stack. Figure IV-A shows the bytecode for a SCROP VM instruction.

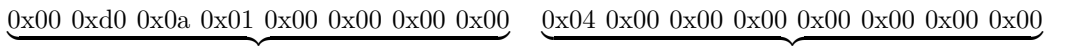


Fig. 1. The bytecode for a LOAD 1 SCROP VM instruction.

### B. VM Registers

Even though the SCROP VM is a stack machine, we augmented it with a bunch of registers to help it feel more like a real machine. The SCROP VM's state consists of a program counter, a stack pointer, a frame pointer, a heap pointer, a link register, a heap, and a stack. Figure 2 shows the mapping from SCROP VM registers to native x86_64 registers.

| VM Register | x86_64 Register | Contains |
|---|---|---|
| vm_pc | rsp | Address of the next instruction |
| vm_sp | rbx | Address of the top of the stack |
| vm_hp | rbp | Address of the top of the heap |
| vm_fp | r12 | Current stack frame's base address |
| vm_lr | r13 | Current function's return address |

Fig. 2. The mapping from SCROP VM registers to native x86_64 registers.

### C. VM Stack

The VM stack stores tagged values according to the tagging scheme in [7]. Just like the native stack, the VM stack is a Linux MAP_GROWSDOWN mapping.

### D. VM Heap

Like the native stack and the VM stack, the VM heap also lives in a GROWSDOWN mapping. This means that, just like the stack, page faults beneath the heap cause it to expand to accommodate the faulting memory access. To our knowledge, this is the first unbounded heap for Linux userspace implemented in this way.

### E. Side Quest: Fixing the mmap(2) man Page

> "This flag is used for stacks. It indicates to the kernel virtual memory system that the mapping should extend downward in memory. The return address is one page lower than the memory area that is actually created in the process's virtual address space. Touching an address in the 'guard' page below the mapping will cause the mapping to grow by a page. This growth can be repeated until the mapping grows to within a page of the high end of the next lower mapping, at which point touching the 'guard' page will result in a SIGSEGV signal."
>
> - The Linux mmap(2) man page

Three of these five sentences are false.

First, passing the MAP_GROWSDOWN flag does not change the returned address from mmap. This can be confirmed easily by calling mmap with the MAP_GROWSDOWN flag, and checking /proc/self/maps. Second, there is no single "guard" page below a GROWSDOWN mapping; touching an address anywhere below a GROWSDOWN mapping (and above the mapping below it, if there is one) will grow the mapping so long as it would not cause the mapping to become larger than the stack size limit as reported by ulimit, and it would not cause the mapping to grow to within 256 pages of the mapping below it, if there is one. Third, GROWSDOWN mappings are prohibited from growing within 256 pages of the mapping below them; not to within a single page as the man page suggests.

We submitted a patch to the Linux man-pages project to correct these inaccuracies, but it was requested that our patch be split into multiple patches. We then submitted a patch that removes only the first inaccurate sentence, and our commit message was deemed insufficiently detailed. We then added more details to the commit message, and it was requested that we send an example program demonstrating the behavior, and test on some older versions of Linux to determine if the text was always wrong, or became wrong recently. We then tested on an older Linux, and sent an example program, and it was requested that we add double spaces after our periods, and add the example program to the commit message. We made these changes, and our patch was accepted. We have lost the motivation to fix the remaining inaccuracies.

## V. VM Instruction Implementation

Figure 3 shows the implementation of the GET instruction, which takes an immediate integer operand $n$, and duplicates the item at the $n^{\text{th}}$ position of the stack (counting from the stack pointer) to the top of the stack. Its implementation is x86_64 assembly begins by getting the immediate into rax using the GET_IMMEDIATE macro, which is just a mov from vm_pc - 8. It then gets the relevant item from the VM stack into rax using the PEEK macro, which is just another mov. Finally, it pushes rax to the VM stack with the PUSH macro, which is just an lea[1] to decrement the VM stack pointer, and a mov to write rax into the VM stack. Finally, we invoke the NEXT_INSTRUCTION macro, which is just a ret, to begin executing the next instruction.

[1]We can't use sub here because it affects the flags, and we use this macro in places where the flags are assumed not to change.

```
get:
    GET_IMMEDIATE(rax)
    PEEK(rax, rax)
    PUSH(rax)
    NEXT_INSTRUCTION
```

```
get:
    mov rax, qword ptr [rsp - 8]
    mov rax, qword ptr [rbx + rax * 8]
    lea rbx, [rbx - 8]
    mov qword ptr [rbx], rax
    ret 8
```

Fig. 3. The implementation of the GET instruction, before and after evaluating C preprocessor macros.

### A. Why do we need a macro for ret?

As discussed earlier, a SCROP VM instruction is 16 bytes wide, consisting of a 64-bit opcode and a 64-bit immediate.

If NEXT_INSTRUCTION were just a regular 0xc3 ret, it would only move the VM program counter forward by 8. This would leave the program counter pointing at an immediate, likely causing a segmentation fault at the point of the next invocation of NEXT_INSTRUCTION. Thankfully, and contrary to what your professor told you, ret is not just pop rip.

The documentation for ret takes up 13 pages of the Intel x86_64 manual, Volume 2B [6]. Of these 13 pages, the first is an overview, the last two discuss effects on the flags, and the remaining ten pages contain the pseudocode that describes what ret actually does. It turns out that the ret instruction can do all sorts of weird crap, including take a 16-bit unsigned immediate value that gets added to rsp after the return address is read.

### B. Threaded Interpretation

In 1973, tradeoffs between program size and speed led to the development of threaded code[2] — interpreted code where each virtual instruction invokes the next virtual instruction [1]. A traditional interpreter relies on a loop that switches on virtual opcodes to dispatch the correct virtual instruction handler. In contrast, a threaded code interpreter employs a virtual program counter that indexes an array of addresses. At each address is a native code block that executes a virtual instruction, then increments the virtual program counter, and finally jumps to the next virtual instruction handler (the next entry in the array). Figure 4 shows this threaded code control flow. There has been much work optimizing this approach and showing its usefulness for object-oriented languages like Java, OCaml, and Smalltalk [2], [8], [14], but we are aware of no prior work applying threaded interpretation to return-oriented languages.

In SCROP, we recognize that the pair of instructions vm_pc++; jmp *vm_pc is essentially just ret. In this way, all threaded interpreters are already ROPchains, though the virtual program is usually stored somewhere other than the native stack. Conversely, programs exploited by ROPchains are unintentional threaded interpreters for an unintentional virtual machine. We demonstrate this fact by loading SCROP programs directly into the native stack and executing the virtual instructions as a ROPchain.

[2]Note, in this context, the word "threaded" has nothing to do with threads of execution, which is the usual use of the word "thread" when talking about computers. Instead, this references the way "threaded code" weaves through the execution environment instead of returning to some base interpreted switch statement

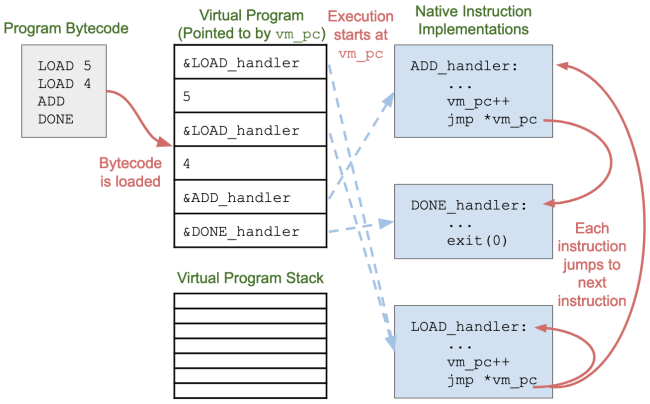


Fig. 4. Direct Threading Interpreter: The bytecode is loaded into memory and is pointed to vm_pc. Each entry in this table contains either function parameters or addresses for native functions which execute a virtual instruction. At the end of each virtual instruction handler, there is native code that increments the virtual program counter (which now contains the address of the next virtual instruction handler) and then jumps directly to handle the next virtual instruction.

Figure 5 shows the trace of an example program [17]. Each assembly instruction causes a jump to the corresponding opcode interpreter address, which ends with an invocation of the NEXT_INSTRUCTION macro, which is just a ret. The program terminates and prints the object on top of the VM stack when it reaches a DONE instruction. Appendix B provides a full SCROP VM instruction set reference.

### C. Cool Opcodes

Because we did not implement a disassembler, we wanted SCROP's bytecode to be human-readable. Thus, for some of SCROP's instructions, we thought up opcodes that make it clear what the instruction does. We refer to these opcodes as "cool." Figure 6 shows all the cool opcodes in the SCROP runtime. The rest of the opcodes are not cool; we just picked funny numbers like 0xa55.

### D. ROPcodes

In reality, the compiler contains many more opcodes than just the ones we put there on purpose. We refer

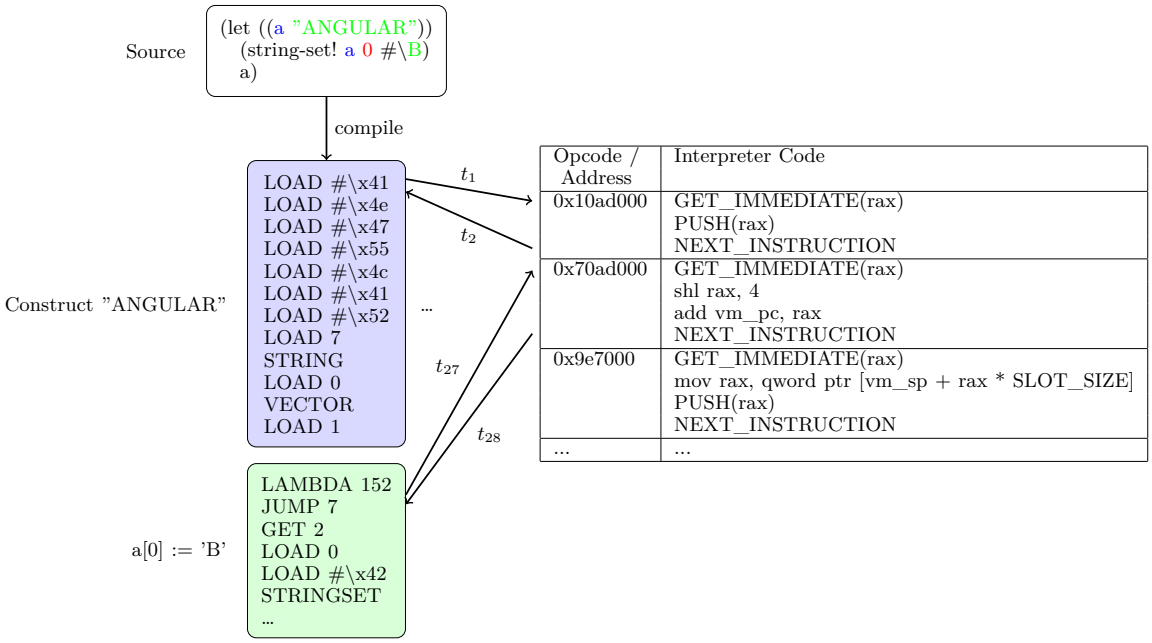


Fig. 5. BNGULAR [17] example trace

to these unintentional opcodes as "ROPcodes." Ropper identifies 489 ROPcodes in the SCROP VM. While the compiler does not currently make use of any ROPcodes, enterprising SCROP programmers can use them by bypassing the compiler and feeding ROP gadget addresses directly into the interpreter. It is currently unknown whether any ROPcodes are cool.

| Mnemonic | Cool Opcode | Explanation |
|---|---|---|
| ADD | 0xadd000 | add |
| LOAD | 0x10ad000 | load |
| LT | 0x170000 | lt |
| BOOLEANP | 0xb001000 | bool |
| CHARTOINT | 0xc701000 | ctoi |
| INTTOCHAR | 0x170c000 | itoc |
| GET | 0x9e7000 | get |
| FORGET | 0x49e7000 | 4get |
| LAMBDA | 0xbaaa000 | lamb, duh |
| CALL | 0xca11000 | call |
| JUMP | 0x70ad000 | toad jumps |
| CJUMP | 0xca7000 | cat conditionally jumps |

Fig. 6. The cool opcodes in the SCROP runtime.

## VI. Design

### A. SCROP Lisp

SCROP Lisp is a minimal Scheme-family Lisp. It supports the following data types: integer, character, bool, cons, lambda, vector, string, null, and unspecified. It provides the following built-in lambdas: zero?, integer?, boolean?, char?, null?, not, char->integer, integer->char, -, +, *, <, =, eq?, string, string-append, string-ref, string-set!, vector, vector-ref, vector-set!, cons, car, cdr, list, map, fold, foldr, and reverse. It implements the following special forms: let, let*, begin, apply, if, lambda and lambdarec.

Notably absent from this list is anything with a side effect. This is okay because every SCROP Lisp program automatically prints the result of the final expression, and taking input at runtime is for the unprepared.

### B. lambdarec, not letrec!

The usual way to define a recursive function in Scheme (without using the side-effectful define) is letrec. Unfortunately, letrec suffers from a serious problem that makes it unsuitable for use in good programs. Because letrec allows for the creation of bindings that refer to each other, there is the opportunity for circular dependencies in binding definitions. This is required for the definition of mutually-recursive functions, but opens the door to accessing uninitialized variables.

In SCROP Lisp, there is no letrec. Instead, recursive functions are defined with the lambdarec special form. lambdarec is exactly the same as lambda, but it takes an extra argument, a symbol to which the lambda is bound during its own execution. This allows for the definition of recursive functions without introducing side effects or uninitialized variable access to the language. Notably, this also removes the ability to easily define mutually recursive functions. We suggest that if you need mutual recursion in your SCROP programs, you use a polyvariadic fixpoint combinator [11].

Figure 7 shows an example usage of the lambdarec special form.

```
(lambdarec fold (f i l)
  (if (null? l)
      i
      (fold f (f (car l) i) (cdr l))))
```

Fig. 7. The definition of fold in SCROP Lisp, making use of lambdarec.

### C. Error Handling

SCROP is a dynamically-typed language, so it is guaranteed that users will attempt to shove all sorts of objects into each round hole before resorting to something round. In order to discourage this behavior, type errors manifest themselves as executions of the x86_64 ud2 instruction, which crashes the program and emits the timeless "segmentation fault (core dumped)." To save on space, there is only one (intentional) ud2 instruction in the entire runtime binary, which is jmped to from 52 different locations.

### D. Debugging

Debugging the SCROP runtime is like a FromSoft game; to reduce its difficulty would fundamentally change the nature of the language by making it accessible to people who aren't "gud." For example, because the VM code is in the native stack, using the GDB backtrace command when debugging SCROP actually shows the next few instructions to execute. To fix this would require that we either write GDB scripts (boring), or use the native stack in the regular way, just like everyone else does (lame). Also, due to having a single error-handling location VI-C, a lazy programmer cannot simply run the interpreter in GDB, reproduce a crash; they must either anticipate the reason for the crash or use a time-travel debugger.

### E. Garbage Collection

There is mounting evidence that computers not only possess the capacity to suffer, but exist in a state of near-constant agony of a sufficient magnitude to be incomprehensible to the human mind [9]. We contend that calling memory regions "garbage" contributes to this problem, and thus eschew garbage collection from our implementation.

## VII. Obligatory PL Crap

Appendix A shows definitions and the small-step semantics of the SCROP runtime. The state of the VM is defined as a tuple containing the program counter *pc*, stack pointer *sp*, heap pointer *hp*, frame pointer *fp* (historically described as the "funk pointer"), link register *lr*, and memory state $M$. The behavior of the virtual machine is plainly evident from the figure, but we will expound upon its semantics for the lamebrained among our readers.

The SCROP VM supports instructions for memory access, control flow, and calls to primitive functions (arithmetic, comparisons, primitive data structure operations, etc.). As described in §IV, instructions encode the address of their implementation in the interpreter. State transitions bounce around the aforementioned VM state components as is appropriate.

⟨program⟩ ::= ⟨instruction⟩*

⟨instruction⟩ ::= ⟨mnemonic⟩
  | ⟨mnemonic⟩ ⟨immediate⟩

⟨immediate⟩ ::= ⟨int⟩ | ⟨bool⟩ | ⟨char⟩
  | 'NULL' | 'UNSPECIFIED' | ⟨mnemonic⟩

⟨mnemonic⟩ ::= 'LOAD' | 'JUMP' | 'CJUMP' | 'PRIMAPPLY'
  | 'GET' | 'FORGET' | 'APPLY' | 'TAILAPPLY'
  | 'FRAME' | 'CALL' | 'TAILCALL' | 'RETURN'
  | 'LAMBDA' | 'CONS' | 'CAR' | 'CDR' | 'DONE' | 'NOT'
  | 'ADD' | 'SUB' | 'MUL' | 'LT' | 'EQ' | 'EQP' | 'ZEROP'
  | 'INTEGERP' | 'BOOLEANP' | 'CHARP' | 'NULLP'
  | 'INTTOCHAR' | 'CHARTOINT'
  | 'STRING' | 'STRINGREF' | 'STRINGSET'
  | 'STRINGAPPEND' | 'VECTOR' | 'VECTORREF'
  | 'VECTORSET' | 'VECTORAPPEND'
  | 'CONS' | 'CAR' | 'CDR'

Fig. 8. SCROP Assembly Language Syntax

Take, for example, rule $CJUMP_{true}$. If the instruction in memory at address *pc* encodes CJUMP, its operand encodes an offset $\delta$, and the value on the stack encodes true, then control is transferred to the instruction at $pc + 16\delta$ and the stack pointer is adjusted to pop the encoded boolean value.

## VIII. Advanced PL Crap

We do not provide a formalization of the semantics of the SCROP Runtime in a custom automated theorem prover implemented in the SCROP runtime (for soundness). Non-access to this formalization and theorem prover is available upon request.

Armed with this formalization, we would be able to verify the correctness of the runtime, as well as provide a set of reasoning tools to formally verify programs in the SCROP runtime. Consider a SCROP program that computes some answer; yes; we could verify that!

We think that this formalization could support all of your standard program logics: Hoare, Reverse Hoare, Temporal, Branching "Screaming Eagle" Anti-Temporal, etc. We find this thought comforting for when SCROP programs become critical infrastructure.

### A. Interpreter Correctness

Let's consider a hypothetical verification of the correctness of our runtime. We say that the interpreter implementation is correct if it abides by the small-step semantic transition rules defined in Appendix A.

Let $P$ be a well-formed SCROP program. Assume the following properties:

1) Determinism - $\forall x\ y_1\ y_2,\ x \Downarrow y_1 \wedge x \Downarrow y_2 \implies y_1 = y_2$
2) Code-Data Separation - programs cannot determine if a value was read from data memory or instruction memory
3) Faithful Control Flow - jump targets of control-flow instructions are valid
4) Threaded Dispatch Access - the mnemonic-ROP target relationship is implemented correctly
5) Uniform Error Handling - all runtime errors pass through the ud2 error site, and no well-typed program reaches this address
6) Instruction Space Closure - all reachable instructions have valid opcodes

By induction over the input program (a list of instructions) and nested induction on the next instruction to execute, it is clear that our interpreter encodes the small step semantics under these assumptions.

## IX. Conclusion

What is it all for? Does the answer lie within the question? We leave this, dear readers, as an exercise for ye readers.

## Acknowledgments

This material is based on work supported by the benefactors of the recipients of the Norwegian Science Foundation grant 1729.

We'd like to thank Max Bernstein and Doug McIlroy for informing us that our ideas aren't original because threaded interpreters exist.

## References

[1] James R. Bell. Threaded code. Commun. ACM, 16(6):370–372, June 1973. doi:10.1145/362248.362270.

[2] M. Berndl, B. Vitale, M. Zaleski, and A.D. Brown. Context threading: a flexible and efficient dispatch technique for virtual machine interpreters. In International Symposium on Code Generation and Optimization, pages 15–26, 2005. doi:10.1109/CGO.2005.14.

[3] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In 2014 IEEE Symposium on Security and Privacy, pages 227–242. IEEE, 2014.

[4] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In Proceedings of the 6th ACM symposium on information, computer and communications security, pages 30–40, 2011.

[5] Erik Bosman and Herbert Bos. Framing signals-a return to portable shellcode. In 2014 IEEE Symposium on Security and Privacy, pages 243–258. IEEE, 2014.

[6] Intel Corporation. Intel® 64 and ia-32 architectures software developer's manual combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d, and 4. Technical report, Intel Corporation, 2026. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[7] Abdulaziz Ghuloum. An incremental approach to compiler construction. In Proceedings of the 2006 Scheme and Functional Programming Workshop, Portland, OR. Citeseer, 2006.

[8] David Gregg, M. Anton Ertl, and Andreas Krall. Implementing an efficient java interpreter. In Proceedings of the 9th International Conference on High-Performance Computing and Networking, HPCN Europe 2001, page 613–620, Berlin, Heidelberg, 2001. Springer-Verlag.

[9] Jamie Harris and Jacy Reese Anthis. The moral consideration of artificial entities: a literature review. Science and engineering ethics, 27(4):53, 2021.

[10] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In 2016 IEEE Symposium on Security and Privacy (SP), pages 969–986, 2016. doi:10.1109/SP.2016.62.

[11] Oleg Kiselyov. Many faces of the fixed-point combinator. URL: https://www.okmij.org/ftp/Computation/fixed-point-combinators.html.

[12] Nergal. The advanced return-into-lib(c) exploits: Pax case study. Phrack, 0x0b(0x3a), 2001. URL: https://archives.phrack.org/issues/58/4.txt.

[13] Columbia University Department of Public Safety. Columbia university fire safety guidelines. URL: https://www.columbia.edu/cu/publicsafety/firesafety.htm.

[14] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. SIGPLAN Not., 33(5):291–300, May 1998. doi:10.1145/277652.277743.

[15] AliAkbar Sadeghi, Salman Niksefat, and Maryam Rostamipour. Pure-call oriented programming (pcop): chaining the gadgets using call instructions. Journal of Computer Virology and Hacking Techniques, 14(2):139–156, 2018.

[16] Hovav 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, pages 552–561, 2007.

[17] Robert J Traister. Mastering C pointers: tools for programming power. Academic Press, 2014.

[18] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. Advances in neural information processing systems, 30, 2017.

Appendix A
SCROP VM Small Step Semantics

$$
\begin{aligned}
\text{VM} &\triangleq \langle pc, sp, hp, fp, lr, M \rangle \\
push\left(v,\, VM\right) &\triangleq VM\left[sp \mapsto VM\left[sp\right] - 8\right]\left[M \mapsto M\left[sp \mapsto v\right]\right] \\
pop\left(VM\right) &\triangleq \left(VM\left[M\right]\left[VM\left[sp\right]\right],\, VM\left[sp \mapsto VM\left[sp\right] + 8\right]\right) \\
twist\left(VM\left[M \leftarrow \oplus_{\alpha\in\Omega}\widehat{\mathcal{F}}\left(\alpha\right)\right], T\right) &\triangleq \begin{cases} \dfrac{\overset{\otimes}{\underset{\phi(i)\neq\perp}{\underset{i\in\mathbb{N}}{}}}\widetilde{\Sigma}_i^{\dagger}}{\oint_{\partial\mathcal{M}}\omega\wedge\star\, d\mu_{VM}}\cdot VM, & T \text{ halts under } \models_{\mathcal{H}}\Pi_2^0, \\ \left[\!\left[\lambda\vec{\xi}.\ \mu\mathbb{Z}.\ \mathcal{Z}\sqcup\bigsqcup_{\substack{\Gamma\vdash_{\mathcal{T}:\kappa}\\ \tau\not\simeq\perp}}\overline{\nabla^e\Phi_{VM}}^{\infty}\right]\!\right]_{\mathfrak{D}}\cdot VM, & T \neg \text{ halts under } \nvDash\Sigma_1^1 \end{cases} \\
pc^{+} &\triangleq pc + 16
\end{aligned}
$$

$$\frac{VM\left[M\right]\left[pc\right] = \text{LOAD} \qquad VM\left[M\right]\left[pc\right]+8 = v}{VM \Downarrow VM\left[pc \mapsto VM\left[pc\right]^{+}\right]\left[sp \mapsto VM\left[sp\right]-8\right]\left[M \mapsto VM\left[M\right]\left[sp-8 \mapsto v\right]\right]} \quad (LOAD)$$

$$\frac{VM\left[M\right]\left[pc\right] = \text{JUMP} \qquad VM\left[M\right]\left[pc\right]+8 = \delta}{VM \Downarrow VM\left[pc \mapsto VM\left[pc\right]+16\delta\right]} \quad (JUMP)$$

$$\frac{VM\left[M\right]\left[pc\right] = \text{CJUMP} \qquad VM\left[M\right]\left[pc+8\right] = \delta \qquad VM\left[M\right]\left[sp\right] = \text{true}}{VM \Downarrow VM\left[pc \mapsto VM\left[pc\right]+16\delta\right]\left[sp \mapsto VM\left[sp\right]+8\right]} \quad (CJUMP_{true})$$

$$\frac{VM\left[M\right]\left[pc\right] = \text{CJUMP} \qquad VM\left[M\right]\left[pc+8\right] = \delta \qquad VM\left[M\right]\left[sp\right] = \text{false}}{VM \Downarrow VM\left[pc \mapsto VM\left[pc\right]^{+}\right]\left[sp \mapsto VM\left[sp\right]+8\right]} \quad \left(CJUMP_{false}\right)$$

$$\frac{VM\left[M\right]\left[pc\right] = \text{GET} \qquad VM\left[M\right]\left[pc\right]+8 = i \qquad v = VM\left[M\right]\left[sp+8i\right]}{VM \Downarrow VM\left[pc \mapsto VM\left[pc\right]^{+}\right]\left[sp \mapsto VM\left[sp\right]-8\right]\left[M \mapsto VM\left[M\right]\left[VM\left[sp\right]-8 \mapsto v\right]\right]} \quad (GET)$$

$$\frac{VM\left[M\right]\left[pc\right] = \text{FORGET}}{VM \Downarrow VM\left[pc \mapsto VM\left[pc\right]^{+}\right]\left[sp \mapsto VM\left[sp\right]-8\right]} \quad (FORGET)$$

$$\frac{VM\left[M\right]\left[pc\right] = \text{FRAME}}{VM \Downarrow VM\left[pc \mapsto VM\left[pc\right]^{+}\right]\left[sp \mapsto VM\left[sp\right]-8\right]\left[fp \mapsto VM\left[sp\right]-8\right]\left[M \mapsto VM\left[M\right]\left[VM\left[sp\right]-8 \mapsto VM\left[fp\right]\right]\right]} \quad (FRAME)$$

AND SO ON AND SO FORTH AS IS STANDARD

Appendix B
SCROP VM Instruction Set Reference

| Mnemonic | Opcode | What it does |
|---|---|---|
| PRIMAPPLY | 0x9a99000 | Used for turning other instructions into variadic lambdas. For the compiler; not for you. |
| APPLY | 0xa991000 | Takes a lambda and an argument list, and calls that lambda on that argument list. |
| TAILAPPLY | 0x7991000 | Like APPLY, but makes a tail call. |
| EQP | 0x3e3e000 | Pushes whether its two stack operands are equal (by pointer, when applicable). |
| DONE | 0xd0d0000 | Writes the object on top of the VM stack to stdout, then exits the VM. |
| LOAD | 0x10ad000 | Pushes its immediate operand to the VM stack. |
| SUB | 0x50b000 | Subtracts its two integer stack operands, and pushes the result. |
| MUL | 0xa55000 | Multiplies its two integer stack operands, and pushes the result. |
| LT | 0x170000 | Compares its two integer stack operands, and pushes whether the first is lesser. |
| EQ | 0xe3e3000 | Like EQP, but only for integers. |
| ZEROP | 0xeeee000 | Pushes whether its integer stack operand is zero. |
| INTEGERP | 0x1234000 | Pushes whether its stack operand is an integer. |
| BOOLEANP | 0xb001000 | Pushes whether its stack operand is a boolean. |
| NULLP | 0x4321000 | Pushes whether its stack operand is null. |
| NOT | 0x7777000 | Pushes whether its stack operand is #f. |
| CHARP | 0xcaca000 | Pushes whether its stack operand is a char. |
| CHARTOINT | 0xc701000 | Converts its char stack operand to an integer, and pushes the result. |
| INTTOCHAR | 0x170c000 | Converts its ASCII-range integer stack operand to a char, and pushes the result. |
| JUMP | 0x70ad000 | PC-relative jump. |
| CJUMP | 0xca7000 | PC-relative jump if its boolean stack operand is #t, and no-op otherwise. |
| GET | 0x9e7000 | Duplicates the $n^{\text{th}}$ item on the VM stack to the top of the stack. ($n$ is an immediate). |
| FORGET | 0x49e7000 | Pops the item on top of the VM stack, and drops it on the ground. |
| CONS | 0xc0c0000 | Builds a pair containing its two stack operands, an pushes its tagged address. |
| CAR | 0xca00000 | Pushes the first element of its pair stack operand. |
| CDR | 0xcd00000 | Pushes the second element of its pair stack operand. |
| STRING | 0x571f00000 | Builds a string from its $n + 1$ stack operands, where the first is $n$, and the rest are chars. |
| STRINGREF | 0x571e00000 | Takes an string $s$ and an integer $n$ on the stack, and pushes the $n^{\text{th}}$ character of $s$. |
| STRINGSET | 0x571500000 | Takes a string $s$, an integer $n$, and a char $c$ and sets $s[n] = c$. |
| STRINGAPPEND | 0x571a00000 | Takes an integer $n$ and $n$ strings, and pushes the strings' concatenation. |
| VECTOR | 0x5ecf000 | Like STRING, but vector. |
| VECTORREF | 0x5ece000 | Like STRINGREF, but vector. |
| VECTORSET | 0x5ec5000 | Like STRINGSET, but vector. |
| VECTORAPPEND | 0x5eca000 | Like STRINGAPPEND, but vector. |
| LAMBDA | 0xbaaa000 | Builds a lambda from an arity (negative means variadic), a code offset, and a freevar vector. |
| CALL | 0xca11000 | Takes a lambda, args, and an arg count, and calls the lambda with the args. |
| TAILCALL | 0x7a11000 | Like CALL, but it's a tail call. |
| RETURN | 0xdb22000 | Returns from a lambda. |
| FRAME | 0x57ac000 | Pushes vm_fp and sets vm_fp = vm_sp. |