VEP: a Virtual Machine for Extended Proof-Carrying Code*

Heidar Pirzadeh
University of Montreal, Montreal, Canada
Pirzades@iro.umontreal.ca

Danny Dubé
Université Laval, Quebec City, Canada
Danny.Dube@ift.ulaval.ca

ABSTRACT

One of the key issues with the practical applicability of Proof-Carrying Code (PCC) and its related methods is the difficulty in communicating the proofs which are inherently large. One way to alleviate this problem is to transmit, instead, a proof generator for the program in question in a generic extended PCC framework (EPCC). The EPCC needs to provide the execution of the proof generator at the consumer side in a secure manner. The ability to securely run arbitrary untrusted proof generator is a challenging problem.

We explore the design of a small and safe virtual machine (VEP) which provides the EPCC with a robust security guarantee. The VEP is a minor TCB extension of less than 300 lines of code which works as a safe execution environment and brings about a practical solution to the common security and resource management issues.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Stack-oriented processors*;
D.2.4 [**Software Engineering**]: SoftwareProgram Verification—*Correctness proofs*;
D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*;
D.3.2 [**Programming Languages**]: Language Classifications—*Macro and assembly languages*;
D.3.4 [**Programming Languages**]: Processors—*Interpreters*

General Terms

Languages, Verification, Security

Keywords

Proof-Carrying Code, Virtual Machine

*This work was supported by the Natural Sciences and Engineering Research Council of Canada.

VMSEC'08, October 31, 2008, Fairfax, Virginia, USA.

1. INTRODUCTION

Proof-Carrying code (PCC) is a static code analysis technique in which the code consumer is enabled to verify that a received code from an untrusted producer complies with its safety policy. The safety policy is specified by means of a set of axioms and rules that the code producer can use for the purpose of constructing a proof. Using the verification condition generator (VCGen), the consumer constructs a verification condition (VC) which is a formula in a certain logic. The VC has the property that it is provable only if the code respects the safety policy. The constructed VC then is sent to the code producer (or the producer, given a copy of the VCGen, can construct the VC himself). The code is accompanied by what the code producer claims to be the proof of the VC. Before executing the code, the consumer uses a proof checker to verify that the received proof is indeed a proof of the VC. If so, the code is safe and can be executed. Figure 1 shows the interaction between the entities involved.

Once the safety of an untrusted code is successfully established, there is no need to check the code anymore. As a result, we have a computing system with less overhead and more security [1, 9].

The traditional PCC approach suffers from some shortcomings. Apart from the difficulty of building or generating the proofs for the code, one of the crucial obstacles for the practical applicability of Proof-Carrying Code and related techniques is the size of the proofs that must accompany the code. It is important to have a compact representation of the proofs because they are possibly sent through communication networks. This difficulty of communicating the proofs, which are inherently large, makes the PCC less scalable. In traditional PCC framework, it is not unusual to see proofs that are 1000 times larger than the associated code, which makes the use of PCC impractical for all but the tiniest examples [10].

Another issue in PCC framework is that it does not provide the producer with enough flexibility. That is, the producer is constrained to submit a proof in a logic which has been imposed by the consumer. That is, even if the producer finds it possible to build a simpler proof in a higher-order logic, he is forced to build the proof in the consumer's logic which might result in an overweight proof.

The anxiety about the trusted computing base (TCB) grows along with its number of lines. Any bug in these components can compromise the security of the whole system. Therefore, to have a safe and implementable PCC framework, one of the obstacles in front is to make a big TCB enlargement.

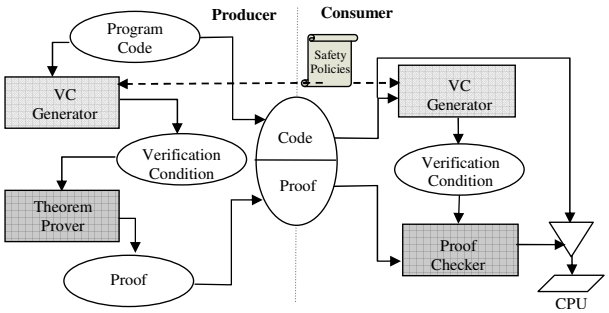

Figure 1: Traditional PCC framework

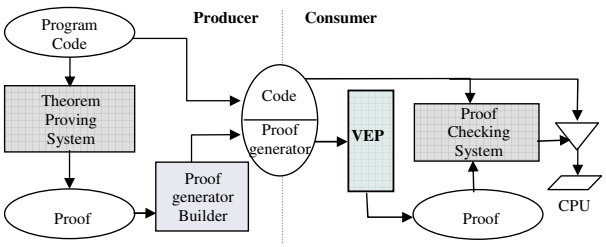

Figure 2: The generic EPCC framework

Extended Proof-Carrying Code (EPCC) tackles these problems by equipping the traditional PCC framework with a small safe virtual machine. The virtual machine of EPCC (VEP), enables the EPCC framework to offer a solution to the PCC's scalability issue. This article, presents the design of the VEP and describes the trade-offs we had to make throughout the design process. It also discusses the way in which VEP works.

In the rest of this article, we first present our generic Extended Proof-Carrying Code framework in Section 2 and present an overview of the requirements for the VEP in Section 3. Sections 4 and 5 give the detailed design process of the VEP. Section 6 discusses the way in which the VEP enforcement security requirements. Section 7 presents a detailed description of the workings of the VEP and, at the same time, demonstrates the simplicity of the VEP. Finally, we briefly discuss about other VMs, present an experiment, present future work, and conclude.

2. EXTENDED PROOF-CARRYING CODE FRAMEWORK

One of the crucial issues for the practical applicability of PCC and its related techniques is the size of the proofs that must accompany the code. Therefore, it is desirable that proofs be represented in a compact format. One way to reach this goal is *proof optimization* in which the proofs are rewritten in a more compact form which preserves the meaning of the proof of the original form [12, 3]. This could be done finding for a given term t a smaller equivalent term s and replacing all the occurrences of t with s in the proof (e.g., in the arithmetic system, there could be a rule $x * 1 \rightarrow x$ which always reduces the size of a term). Using proof optimization in an approach called lemma extraction, Necula *et al.* could not obtain a reduction better than 15% in the size of the proofs.

Another way of compacting the proofs is through *data compression*. Data compression techniques try to find more compact representations for data, from which the original data can be reconstructed exactly. Many such algorithms compress data by searching for more efficient encodings that take advantage of repetition in the data. These techniques are not well exploited in PCC framework due to the following reasons. The consumer of compressed data must first decompress it, this needs a safe decompresser on the consumer side. Generating the proof of safety for a normal decompressor (relatively big program with about 3000 lines of code) is a difficult task not worth performing because such a decompressor would be a specific decompresser that cannot have the potential to work with a proof compressed by an appropriate but different compressor. That is, to gain the advantage of a good compression, each time, the safety of a new decompresser should be proven according to the compression method which is appropriate for the safety proof of a code. In OPCC approach, Necula *et al.* used the idea of the proof compression. Although their approach resulted in proofs which were smaller than the original proofs, they payed the price of a considerable enlargement of the TCB [10, 13, 8]. We are not in favor of compromising the security of the system with a big TCB expansion simply because the proofs are too large.

We present an extended framework that allows the PCC proofs to be represented as programs. This contributes to reduce the negative impact of the size of the proof and enables the PCC to handle even very large programs. The idea of representing the proofs as programs is inspired by the *Kolmogorov complexity*. Roughly speaking, the Kolmogorov complexity of a string x is the shortest computer program that produces x, i.e., that computes it, prints it, and then halts. One important observation is that this measure of complexity indicates how much a string (or, in the context of proof-carrying code, a proof) can be compressed: the ideal compressed form for a given proof is the shortest program that outputs that proof. Formally, the Kolmogorov complexity $K_{\mathsf{U}}(x)$ of a string x is defined as the length of the shortest program capable of producing x on a universal computer U (such as a Turing machine). $K_{\mathsf{U}}(x) = \min_{p \in \{0,1\}^*}\{\ell(p) \colon p \text{ on } \mathsf{U} \text{ outputs } x\}$. Note that the Kolmogorov complexity is incomputable.

The idea behind the Extended Proof-Carrying Code (EPCC) [11] is simply to send the proof in the form of a program. In this way, we make it possible for the producer to send a proof generator instead of the proof where, according to Kolmogorov complexity, the proof generator ideally can be the shortest program which can output the original proof. For this to work, the consumer should be capable of running the proof generator on a universal computer, in a secure manner, and obtain the proof.

A diagram of an EPCC system is given in Figure 2. In an EPCC system, there are two main parties, a code producer, who sends a code along with its safety proof generator, on the left-hand side, and a code consumer, who wishes to run the code, provided that it is proven safe by the system, on the right-hand side.

The communication between these two parties may consist of a multi-step interaction between the producer and the consumer depending on the underlying proof-carrying code framework that they extend. Generally, at the first step, the producer runs a theorem prover to get a safety proof of

the code he intends to send. Here, in contrast with other PCC frameworks, the consumer is not forced to generate the safety proof in the logic that the consumer imposes. The producer can use this opportunity to build the proof in a logic (e.g., a higher-order logic) that results in a smaller proof. In other words, the producer has the possibility of reducing the size of the safety proof by using a custom logic which can be later converted (translated) to the logic set by the consumer.

Then, the producer builds a proof generator. In accordance with the Kolmogorov complexity, this proof generator can, in principle, be the shortest program which can output the safety proof in the format which is acceptable to the consumer. That is to say, the generic EPCC framework provides the producer with the opportunity of compacting the proof in two steps of optimization and compression.

In the next step, the producer submits the code accompanied by its safety proof generator to the consumer. The consumer is required to check the proof before executing the code submitted by the producer. Therefore, he runs the safety proof generator on the virtual machine of EPCC (the VEP) and obtains the safety proof. Then he runs the proof checker. Upon success, the consumer can repeatedly execute the code safely. As one can easily observe, the EPCC framework is tamper proof, like PCC.

One of the crucial components in the EPCC framework is the VEP which is a universal computer in the trusted computing base of the EPCC. The safe execution of the proof generator depends on the safety of the VEP and the way it imposes the security requirements.

3. CAPTURED REQUIREMENTS FOR THE VEP

In EPCC framework, we execute the proof generator on the VEP. The proof generator can be a package of a decompression algorithm and the compressed proof. In this way, by executing the proof generator, the consumer is actually decompressing the compressed proof. We used the GUNZip algorithm as a representative of algorithms within the decompression techniques area. As a guideline, we tried to design the VEP in a way that it can support an efficient execution of programs written in a broad range of languages.

The virtual machine design process starts by capturing the requirements. In the case of VEP, we dealt with the following requirements.

1. VEP should provide us with a platform which has the potential of working with the Kolmogorov ideal compressor. According to the Kolmogorov complexity, this ideal compressor runs on a universal computer.

2. It should enable the execution of the proof generator at the consumer side in a secure manner. That is, VEP should provide a tightly controlled set of resources for proof generator. Network access, the ability to inspect the host system, or read from input devices and write into file streams should be disallowed. Therefore, VEP should be able to perform execution monitoring.

3. As indicated in EPCC framework, VEP is a part of the TCB. Knowing that any bug in TCB can compromise the security of the whole system, we need the VEP to have a size and simplicity, such that, it be feasible for a human to inspect and verify it by pen and paper. This would give the VEP the potential to be proved safe by the PCC itself.

4. The proof generators are sent in the language of VEP. Consequently, the language of VEP is a factor which can affect the size of proof generator. It would be helpful if VEP can provide us with small size of code.

5. In EPCC framework the producer sends a proof generator to the consumer side. The consumer should run the proof generator on his side to obtain the proof. Considering that any execution has an overhead, code execution performance is a concern.

6. A virtual machine can die a quick death because no one writes codes specific to that VM. This could be due to its high complexity. Therefore, we want the complexity of VEP to be low, and design it in a way that has the potential to become popular.

The mentioned goals and requirements are not equally important to us. The three first items of the above list are of very high importance. There also exist trade-offs between the list items. For instance, the low complexity and small code size, both depend on the number of instructions in VEP instruction set; having small set of instructions results in a virtual machine with low complexity and, on the other hand, a big list of instructions makes the code smaller. Although these two factors are contradictory, there can be a good balance between them. As a result, finding a good trade-off is our major endeavor. For this, in making a design trade-off, we favor the more important over the less important cases. Having established the set of requirements, we discuss the design choices in the following.

4. MACHINE TYPE

Conventionally, a VM can either be stack-based or register-based. In order to decide between these two design choices, we kept an eye on the captured requirements. Implementing a universal computer can be done with a stack machine which has more than one stacks or has one stack with random access. Nevertheless, register machines can be universal computers, therefore, both approaches can satisfy the requirement 1.

The most popular virtual machines, like Java Virtual Machine [6] and Common Language Runtime [7], use a stack machine type rather than the register-oriented architectures used in real processors, due to the simplicity of their implementation. Hence, a stack-based machine helps us to better fulfill the requirement 3 on the list. Furthermore, the simple stack operations can be used to implement the evaluation of any arithmetic or logical expression and any program written in any programming language can be translated into an equivalent stack machine program. Moreover, the stack machines are easier to compile to. Potentially, this could prevent a quick death of the VEP to happen as stated on the requirement 6.

The last among the reasons which led us to choose the stack machine type over the register one was the fact that a compiled code for a stack machine has more density than the one for the register machine. In an experiment, Davis et al. [4] the corresponding register format code after elimination of unnecessary instructions was around 45% larger

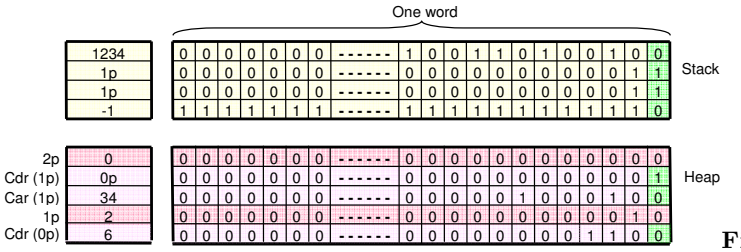

Figure 3: Schemata of the stack and the heap

than the stack code needed to perform the same computation. This can specially affect the size of the proof generator written for VEP as mentioned in requirement 4. Accordingly we chose the stack machine type over the register one.

5. INSTRUCTION SET ARCHITECTURE

The *Instruction Set Architecture* (ISA) of a virtual machine is the VM interface to the programmer. In the case of VEP, available data types, the set of memory spaces are defined by ISA. The ISA definition also includes a specification of the set of opcodes (machine language) and the VEP's instruction set. Next, we discuss each of these parts and their design choices.

5.1 Data Types

On the VEP, we have two distinct types of values: *numbers* and *pairs*. Considering that the VEP is implemented using 32-bits machine words, the least significant bit of the cell shows the data type of the stored value in that cell. This bit is not visible to the programmer while the remaining 31 bits are visible. If we have a cell that references a *pair*, the contents of the cell represents the address of a pair in the heap memory. For a cell with its type *number*, the contents of the cell is a signed integers.

5.2 Memory

The VEP uses three blocks of memory: a code space (whose cells are bytes), a heap (made to contain pairs of machine words), and a stack (whose cells are machine words). The stack grows towards the high addresses (the first item pushed on the stack is stored at address zero) and the stack pointer points at the topmost element. The heap provides the programmers with additional flexibility by supplying the VEP with memory for objects of arbitrary lifespan.

Figure 3 shows schemata of the stack, the heap, and the code space in VEP. For each of these three schemata, sample binary content are shown on the right-hand side and the human readable format of the same content on the left-hand side. The second stack element from the top has the type *pair* (the type bit is one) and rest of the bits show the address of the pair in the heap which is 1 (1p in human-readable format). The pair 1p in the heap is a pair of the two values 34 and 0p which are respectively the *car* and the *cdr*[1] of 1p, where *car* returns the first item of the pair and *cdr* returns the second one. It should be mentioned that the values in the pairs follow the same typing convention as we have in the stack.

[1]Generalized from the LISP operations on binary tree structures, where *cdr* returns a list consisting of all but the first element of its argument and *car* returns the first element.

Rank	80x86 instructions	% Execution
1	Data transfer instructions	38.00%
2	Control instructions	22.00%
3	Comparison instructions	16.00%
4	Arithmetical instructions	13.00%
5	Logical instructions	6.00%
	Total	96.00%

Figure 4: Instruction distribution approximation

5.2.1 Memory Management

The VEP provides automatic memory management of the heap, thus there can be no dangling reference or memory leak due to manual memory management errors and the programmer can put more time on productivity instead of managing low-level memory operations.

Reference counting is the automatic memory management technique used in VEP. In reference counting technique, each object (pair) in the heap contains a counter which tracks the number of references to that object (references from stack elements andor heap objects). The reference-count field of the object is incremented when there is a new reference to that object, and it is decremented when the reference is removed. The VEP uses a word-sized reference counter which is as large as the maximum number of references the memory can hold. In Figure 3, the content of the address 1p, which is not visible to the programmer, is the number of references to the pair 1p.

When the reference count falls to zero, there are no more references to the object. Therefore, we can immediately find unreachable objects, and then reclaim them. When the reference count falls to zero, the reference counts of offspring objects should be decremented before the object is reclaimed.

A major drawback of reference counting is its failure in reclaiming cyclic garbage data structures. Every value in the VEP is built up out of existing values, hence, it is impossible to create a cycle of references, resulting in a reference graph (a graph which has edges from objects to the objects they reference) that is a directed acyclic graph. Therefore, the reference counting in the VEP does not have the drawback of failure in reclaiming cyclic data structures.

5.3 Instruction Set

The design of the instruction set is one of the most interesting and important aspects of VEP design. The code space, being made of bytes, naturally leads to an instruction set of 256 instructions. The VEP has a RISC-like instruction set which provides random access to stack, plenty of arithmetic, logical, comparison, data transfer, and control instructions and restricted access to the pair-based heap. This gives application developers a good flexibility in implementing their ideas and innovations. It also guarantees an acceptable execution performance (requirement 5). We provide the VEP with a rich set of data transfer instructions which might help to execute the proof generators on the VEP more efficiently.

The VEP instructions can be classified into the following categories.

- **Data transfer instructions** (POP, PEEK, POKE, LOAD1, LOAD2, LOAD3, LOAD4, PEEKI *n*, POKEI *n*, LOADI *n*, PUSH-PC, READC): these instructions

move data from one location in memory to another. These instructions come in a variety of ranges and density of operations, for instance, PEEKI *n*, POKEI *n* have shorter range (i.e. they perform their operations on the eight top elements of the stack), while PEEK and POKE have broader range and less density of operations (e.g. a LOAD1 -1 followed by a PEEK, is equivalent to PEEKI *-1*).

- **Control instructions** (HALT, NOP, JUMP, JMPR, JMPRF, JMPRT): machines and processors, by default, work on instruction sequence. Redirection from this sequence is possible through control instructions. The most basic and common kinds of program control are the unconditional jump and the conditional jumps (branches). Control instructions also include instructions which directly affect the entire machine such as HALT or *no operation* (NOP).
- **Comparison instructions** (EQU, LEQ, LTH, NEQ): the compare instructions compare values by using a specific comparison operation. Typical logical operations include equal and not equal.
- **Arithmetic instructions** (ADD, SUB, MUL, DIV, MOD): the basic four integer arithmetic operations are addition, subtraction, multiplication, and division.
- **Logical instructions** (BSHIFT, BAND, BNOT, BOR): these instructions usually work on a bit by bit basis. Typical logical operations include logical negation or logical complement, logical and, logical or.
- **Heap related instructions** (CONS, CAR, CDR, ISPAIR): these instructions whether perform their action on a pair (CAR and CDR, respectively return the first and the second item of a pair), result in a pair (CONS), or verify if an stack element is a pair (ISPAIR).
- **Input/Output instructions** (OUTPUT): the VEP provides a tightly-controlled set of resources for proof generators to run in. In order to be able to output the resulting proof, a proof generator is allowed to print characters onto the standard output. This is the sole way provided by the VEP for a proof generator to communicate with the outside world. Other than that, network access, the ability to inspect the host system, or reading from input devices and writing into file streams are disallowed. In this sense, the VEP performs the dynamic analysis.

The distribution of the instructions is based on an approximate interpretation of the work of Hennessy *et al* [5] in which 10 simple instructions that account for 96% of instructions executed for a collection of integer programs running on the popular Intel 80x86 was presented (Figure 4). We also kept an eye on our representative algorithm to find out how frequently some operations are executed.

6. SECURITY ENFORCEMENT BY THE VEP

We designed the VEP such that it guarantees a certain number of fundamental safety properties in order to execute the untrusted code in a secure manner. *Memory safety* is one of these properties which prevents reading and writing to illegal memory locations. The code space is read-only and the legal code space locations are $0, \ldots, N_c - 1$, where N_c is the code size. Even the instruction loading must be performed as legal reads from the code space.

In the case of the stack, reads and writes are permitted. Any read or write to the stack is preceded by a memory check which ensures that the read and write are going to be performed on valid stack locations as their destination. What a valid destination is varies from instruction to instruction. Generally, the valid read and write destinations are stack locations from the bottom of the tack to the top of the stack.

In the case of the heap, reads and writes are very restricted. Since the construction of the pairs is governed by the VEP, the programmer has no means to modify the type bit to *forge* a new pair and he has no means to read and write in the heap other than to use CONS, CAR, CDR. Furthermore, memory safety in the VEP asserts that each operation has a sufficient amount of required memory (stack and/or heap) to perform the instruction (e.g., the VEP raises an error if an attempt is made to pop when the stack is empty or to push an item onto a full stack). *Control-flow safety* prevents jumps outside of the code space and *resource bound check* enforces limitations on the size of the code space, the size of the stack, the size of the heap, and the number of instructions the VEP may execute. There are other security requirements such as *type safety* and *numeric safety* which will be explained in following subsections. It should be mentioned that the VEP has the ability to deal with all errors automatically.

The security enforcement by the VEP is simple and straightforward. The VEP enforces these security requirements at different levels. Categorizing the security checks according to their enforcement level shows better how easy the VEP security enforcement is to perform and understand (interested readers can find a complete schema of the security enforcement mechanism in Appendix A).

6.1 Initial Security Enforcement

The VEP checks the following requests for resources, only once, just before executing the code. Note that each request is made using a declaration in the header of the untrusted code. Each time, the VEP verifies whether the demanded amount of resources is no greater than the maximum value settled in an agreement between the producer and the consumer. The demanded code size and demanded stack size are, respectively, denoted by N_c and N_s. The amount of demanded heap size of the untrusted code, is represented in number of pairs N_h.

- *Code size*: if the VEP refuses or fails to allocate the requested block of memory, the VEP refuses the untrusted code. Otherwise, the VEP allocates a block of N_c bytes of memory as the code space and inserts the code into the code space.
- *Stack size*: if the VEP refuses or fails to allocate the requested block of memory above agreed-upon limit, the VEP refuses the untrusted code. Otherwise, the VEP allocates a block of N_s words of memory as the stack memory.
- *Heap size*: if the VEP refuses or fails to allocate the requested block of memory, the VEP refuses the proof

	LOAD1	LOAD2	LOAD3	LOAD4	PEEK	POKE	LOADI n	PEEKI n	POKEI n	POP	PUSH-PC	READC	HALT	JUMP	JMPR	JMPRF	JMPRT	NOP	EQU	LEQ	LTH	NEQ	ADD	SUB	MUL	DIV	MOD	BAND	BOR	BNOT	BSHIFT	CONS	CAR	CDR	ISPAIR	OUTPUT
U					√	√			√	√		√		√	√	√	√		√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
T					√	√						√		√	√	√	√			√	√		√	√	√	√	√	√	√	√	√		√	√		√
R																										√	√									√
C	√	√	√	√								√																								
S					√	√		√	√																											
O	√	√	√	√			√	√			√																					√				
H																																				

U: Stack underflow check **T**: Type of operands check **C**: Code space access check **O**: Stack overflow check
R: Range of operands check **S**: Stack space access check **H**: Heap overflow check

Figure 5: Instruction-wise security enforcement

generator. Otherwise, the VEP allocates a block of N_h words of memory as the heap memory.

- *Timeout*: the untrusted code should finish its task within a definite time period (i.e. number of operations) (N_o). In the case where the N_o is more than the limit the VEP refuses the proof generator.

When the code is not refused during the initial security enforcement, it is ready to be executed by the VEP.

6.2 Global Security Enforcement

Throughout the execution, the VEP enforces two security checks globally. That is, these two checks are independent of the actual next instruction that is about to be executed. The global security enforcement consists of checking the followings.

- *Execution time*: before fetching the next instruction, the VEP makes sure that the code execution time (measured as the number executed operations) has not exceeded the N_o. If the number of executed operations is less than the approved number, then the check is passed, otherwise the code is refused for having run for too long.

- *Program counter*: the VEP should check if the program counter points inside the code space (i.e., non-negative and less than the code size).

6.3 Instruction-wise Security Enforcement

The third level of security enforcement by the VEP is the fine-grained level and is done per instruction. By enforcing this level of security checks the untrusted code is prevented to perform any unsafe operation.

Generally, after fetching each instruction and before the execution of the instruction, the VEP performs a combination of the following checks.

- *Number of operands*: the number of operands of an instruction can vary from zero to two implicit operands on the stack, depending on the instruction. For an instruction that requires with one or more operands on the stack, the existence of a sufficient number of operands must be checked before execution of the instruction. If insufficient operands lie on the stack, the execution is discontinued and the untrusted code gets refused.

- *Type of operand*: the VEP checks if the type of the operands conforms with the operation. As mentioned earlier, the values in the VEP can be numbers or pairs. The VEP can distinguish the type of an operand according to its type bit. Depending on the instruction and the operand, the latter may have to be a number, it may have to be a pair, or it may be free to be of either types. Checking the type of operands ensures that a code is well-typed according to the VEP's type system. That is, the operations are applied only to operands with correct types.

- *Legal range of operands*: the arithmetic instructions should have legal arguments. The VEP checks the operand legality to prevent potential error of using partial operators with arguments outside their defined domain (e.g., division by zero).

- *Legal code destination*: before changing the program counter to the jump destination, the VEP checks if the destination is within the code space. It should be mentioned that the VEP does not enforce the concept of instruction boundaries.

- *Legal stack destination*: For any instruction which results in a read or write to the stack, the VEP ensures that the reads and writes have legal stack locations as their destination.

- *Stack overflow*: the VEP verifies whether there is enough stack to perform an instruction which works with stack memory.

- *Heap overflow*: the VEP verifies whether there is enough free space on the heap to perform an instruction which works with the heap memory.

As it is shown in Figure 5, the complete set of instructions with their safety checks can be simply put into a table. In this way it would be an easy task to verify the safety of the VEP by pen and paper.

7. SEMANTICS OF THE PROGRAMS FOR THE VEP

We start with some notation. We denote *sequences* (or arrays) as comma-separated elements between angles; e.g., $\langle 4, 2, 98\rangle$. When we are interested in referring to the last few elements of a sequence, we write for instance $\langle \sigma | x, y, z\rangle$ to indicate that the sequence is a (sub-)sequence σ followed by

Instr.	$\mathcal{S}$ before	Saf. check	PC after	$\mathcal{S}$ after	$\mathcal{H}$ after	O	where ...
LOAD1	$\langle \mathcal{S}' \vert \rangle$	$T_c(\text{PC}+1)$	PC $+2$	$\langle \mathcal{S}' \vert n \rangle$	$\mathcal{H}$	ϵ	$n = F_l(\mathcal{C}[\text{PC}+1], 0, 0, 0, 24)$
LOAD2	$\langle \mathcal{S}' \vert \rangle$	$T_c(\text{PC}+2)$	PC $+3$	$\langle \mathcal{S}' \vert n \rangle$	$\mathcal{H}$	ϵ	$n = F_l(\mathcal{C}[\text{PC}+1], \mathcal{C}[\text{PC}+2], 0, 0, 16)$
LOAD3	$\langle \mathcal{S}' \vert \rangle$	$T_c(\text{PC}+3)$	PC $+4$	$\langle \mathcal{S}' \vert n \rangle$	$\mathcal{H}$	ϵ	$n = F_l(\mathcal{C}[\text{PC}+1], \ldots, \mathcal{C}[\text{PC}+3], 0, 8)$
LOAD4	$\langle \mathcal{S}' \vert \rangle$	$T_c(\text{PC}+4)$	PC $+5$	$\langle \mathcal{S}' \vert n \rangle$	$\mathcal{H}$	ϵ	$n = F_l(\mathcal{C}[\text{PC}+1], \ldots, \mathcal{C}[\text{PC}+4], 0)$
PEEK	$\langle \mathcal{S}' \vert n \rangle$	$T_s(\mathcal{S}, 0, n)$	PC $+1$	$\langle \mathcal{S}' \vert v \rangle$	$\mathcal{H} + v$	ϵ	$v = F_s(\mathcal{S}, 0, n)$
POKE	$\langle \mathcal{S}' \vert v_1, n \rangle$	$T_s(\mathcal{S}, 0, n)$	PC $+1$	$\langle \mathcal{S}'' \vert \rangle$	$\mathcal{H} - F_s(\mathcal{S}, 0, n) - v_2$	ϵ	$\langle \mathcal{S}'' \vert v_1, v_2 \rangle = F_u(\mathcal{S}, 0, n, v_1)$
LOADI n	$\langle \mathcal{S}' \vert \rangle$		PC $+1$	$\langle \mathcal{S}' \vert n \rangle$	$\mathcal{H}$	ϵ	
PEEKI n	$\langle \mathcal{S}' \vert \rangle$	$T_s(\mathcal{S}, 1, n)$	PC $+1$	$\langle \mathcal{S}' \vert v \rangle$	$\mathcal{H} + v$	ϵ	$v = F_s(\mathcal{S}, 1, n)$
POKEI n	$\langle \mathcal{S}' \vert v_1 \rangle$	$T_s(\mathcal{S}, 1, n)$	PC $+1$	$\langle \mathcal{S}'' \vert \rangle$	$\mathcal{H} - F_s(\mathcal{S}, 1, n)$	ϵ	$\langle \mathcal{S}'' \vert v_2 \rangle = F_u(\mathcal{S}, 1, n, v_1)$
POP	$\langle \mathcal{S}' \vert v \rangle$		PC $+1$	$\langle \mathcal{S}' \vert \rangle$	$\mathcal{H} - v$	ϵ	
PUSH-PC	$\langle \mathcal{S}' \vert \rangle$		PC $+1$	$\langle \mathcal{S}' \vert n \rangle$	$\mathcal{H}$	ϵ	$n =$ PC
READC	$\langle \mathcal{S}' \vert n_1 \rangle$	$T_c(n_1)$	PC $+1$	$\langle \mathcal{S}' \vert n_2 \rangle$	$\mathcal{H}$	ϵ	$n_2 = \mathcal{C}[n_1]$
HALT	$\langle \mathcal{S}' \vert \rangle$		—	—	—	ϵ	
JUMP	$\langle \mathcal{S}' \vert n \rangle$		n	$\langle \mathcal{S}' \vert \rangle$	$\mathcal{H}$	ϵ	
JMPR	$\langle \mathcal{S}' \vert n \rangle$		PC $+ n$	$\langle \mathcal{S}' \vert \rangle$	$\mathcal{H}$	ϵ	
JMPRF	$\langle \mathcal{S}' \vert n, v \rangle$		$F_f(\text{PC}, v, n)$	$\langle \mathcal{S}' \vert \rangle$	$\mathcal{H} - v$	ϵ	
JMPRT	$\langle \mathcal{S}' \vert n, v \rangle$		$F_t(\text{PC}, v, n)$	$\langle \mathcal{S}' \vert \rangle$	$\mathcal{H} - v$	ϵ	
NOP	$\langle \mathcal{S}' \vert \rangle$		PC $+1$	$\langle \mathcal{S}' \vert \rangle$	$\mathcal{H}$	ϵ	
EQU	$\langle \mathcal{S}' \vert v_1, v_2 \rangle$		PC $+1$	$\langle \mathcal{S}' \vert n \rangle$	$\mathcal{H} - v_1 - v_2$	ϵ	$n = ((v_1 = v_2)\ ?\ 1\ :\ 0)$
LEQ	$\langle \mathcal{S}' \vert n_1, n_2 \rangle$		PC $+1$	$\langle \mathcal{S}' \vert n_3 \rangle$	$\mathcal{H}$	ϵ	$n_3 = ((n_1 \leq n_2)\ ?\ 1\ :\ 0)$
LTH	$\langle \mathcal{S}' \vert n_1, n_2 \rangle$		PC $+1$	$\langle \mathcal{S}' \vert n_3 \rangle$	$\mathcal{H}$	ϵ	$n_3 = ((n_1 < n_2)\ ?\ 1\ :\ 0)$
NEQ	$\langle \mathcal{S}' \vert v_1, v_2 \rangle$		PC $+1$	$\langle \mathcal{S}' \vert n \rangle$	$\mathcal{H} - v_1 - v_2$	ϵ	$n = ((v_1 \neq v_2)\ ?\ 1\ :\ 0)$
ADD	$\langle \mathcal{S}' \vert n_1, n_2 \rangle$		PC $+1$	$\langle \mathcal{S}' \vert n_3 \rangle$	$\mathcal{H}$	ϵ	$n_3 = F_r(n_1 + n_2)$
SUB	$\langle \mathcal{S}' \vert n_1, n_2 \rangle$		PC $+1$	$\langle \mathcal{S}' \vert n_3 \rangle$	$\mathcal{H}$	ϵ	$n_3 = F_r(n_1 - n_2)$
MUL	$\langle \mathcal{S}' \vert n_1, n_2 \rangle$		PC $+1$	$\langle \mathcal{S}' \vert n_3 \rangle$	$\mathcal{H}$	ϵ	$n_3 = F_r(n_1 * n_2)$
DIV	$\langle \mathcal{S}' \vert n_1, n_2 \rangle$	$T_z(n_2)$	PC $+1$	$\langle \mathcal{S}' \vert n_3 \rangle$	$\mathcal{H}$	ϵ	$n_3 = F_r(\lfloor n_1 / n_2 \rfloor_0)$
MOD	$\langle \mathcal{S}' \vert n_1, n_2 \rangle$	$T_z(n_2)$	PC $+1$	$\langle \mathcal{S}' \vert n_3 \rangle$	$\mathcal{H}$	ϵ	$n_3 = F_r(n_1 \bmod n_2)$
BAND	$\langle \mathcal{S}' \vert n_1, n_2 \rangle$		PC $+1$	$\langle \mathcal{S}' \vert n_3 \rangle$	$\mathcal{H}$	ϵ	$n_3 = F_r(n_1 \ \&\ n_2)$
BOR	$\langle \mathcal{S}' \vert n_1, n_2 \rangle$		PC $+1$	$\langle \mathcal{S}' \vert n_3 \rangle$	$\mathcal{H}$	ϵ	$n_3 = F_r(n_1 \ \vert\ n_2)$
BNOT	$\langle \mathcal{S}' \vert n_1 \rangle$		PC $+1$	$\langle \mathcal{S}' \vert n_2 \rangle$	$\mathcal{H}$	ϵ	$n_2 = F_r(\tilde{}\ n_1)$
BSHIFT	$\langle \mathcal{S}' \vert n_1, n_2 \rangle$		PC $+1$	$\langle \mathcal{S}' \vert n_3 \rangle$	$\mathcal{H}$	ϵ	$n_3 = F_r(\lfloor n_1 * 2^{n_2} \rfloor_{-\infty})$
CONS	$\langle \mathcal{S}' \vert v_1, v_2 \rangle$	$T_h(\mathcal{H})$	PC $+1$	$\langle \mathcal{S}' \vert p \rangle$	$F_p(\mathcal{H}, v_1, v_2) + p$	ϵ	$p = F_n(\mathcal{H})\mathsf{p}$
CAR	$\langle \mathcal{S}' \vert p \rangle$		PC $+1$	$\langle \mathcal{S}' \vert v \rangle$	$\mathcal{H} + v - p$	ϵ	$v = F_1(\mathcal{H}, p)$
CDR	$\langle \mathcal{S}' \vert p \rangle$		PC $+1$	$\langle \mathcal{S}' \vert v \rangle$	$\mathcal{H} + v - p$	ϵ	$v = F_2(\mathcal{H}, p)$
ISPAIR	$\langle \mathcal{S}' \vert v \rangle$		PC $+1$	$\langle \mathcal{S}' \vert n \rangle$	$\mathcal{H} - v$	ϵ	$n = ((\exists p.\ v = p)\ ?\ 1\ :\ 0)$
OUTPUT	$\langle \mathcal{S}' \vert n \rangle$	$T_b(n)$	PC $+1$	$\langle \mathcal{S}' \vert \rangle$	$\mathcal{H}$	n	

Figure 6: Semantics of the instructions

elements x, y, and z. It is possible to read and update a sequence in random-access fashion. The expression $\sigma[i]$ gives the element of σ at index i and $\sigma[i \mapsto v]$ produces a sequence that is identical to σ except that the element at index i is now v. Indices start at 0. The length of σ is $|\sigma|$. The code space is denoted by $\mathcal{C}$, the stack, by $\mathcal{S}$, and the heap, by $\mathcal{H}$. The program counter, which points at the instruction that is being executed, is denoted by PC. $\mathcal{C}$ is a never-updated sequence of bytes. $\mathcal{S}$ is a sequence of machine words that gets updated and whose length varies. $\mathcal{H}$ is a couple (n, H), where H is a sequence of N_h pairs and n is the index of the first free pair in the chain of free pairs. A pair is a triple (r, v_1, v_2) of machine words, where r is the reference count. All the free pairs of a heap are chained together through their third field and the last pair of the chain contains the index N_h, which points beyond the upper bound of H and marks the end of the chain. In the presentation of the semantics, n ranges over values of type *number*, p ranges over values of type *pair*, v ranges over values of either type, and r ranges over reference counts. The semantics makes use of test functions to enforce safety. The name of each test function is T with a subscript; e.g., T_s. Note that failure to satisfy the condition specified by a test function causes the abortion of the execution of the program. The semantics also uses helper functions, whose names are F with subscripts; e.g., F_t.

A program for the VEP contains the four declarations of resource consumption, N_c, N_s, N_h, and N_o, followed by the N_c bytes that make up the code proper. If the VEP accepts the four declarations, it loads the N_c bytes into $\mathcal{C}$. Then, it initializes PC to 0, $\mathcal{S}$ to $\langle\rangle$, and $\mathcal{H}$ to the heap with all pairs in the chain of free pairs:

$$(0,\ \langle (0,0,1), (0,0,2), \ldots, (0,0,N_h) \rangle).$$

After the initialization, the execution of the instructions begins. Before the execution of an instruction, two instruction-independent checks are performed. First, the VEP verifies whether the number of instructions until now is below N_o. If not, the execution is aborted. Second, a check is performed on PC to ensure it points inside of $\mathcal{C}$'s bounds so that loading the next instruction's opcode is safe. That is, the check $T_c(\text{PC})$ is performed (see Figure 8). Once the instruction-independent checks are passed, the opcode of the current instruction, $\mathcal{C}[\text{PC}]$, is read. Depending on the opcode, the execution continues with additional instruction-specific checks, if any, and with the intended effects of the instruction. The details of the execution of each instruction

$$
\begin{array}{rcl}
F_1(\mathcal{H},p) & \triangleq & \textbf{let } (n_1,H)=\mathcal{H} \textbf{ and } n_2\mathtt{p}=p \textbf{ and } (r,v_1,v_2)=H[n_2] \textbf{ in } v_1 \\
F_2(\mathcal{H},p) & \triangleq & \textbf{let } (n_1,H)=\mathcal{H} \textbf{ and } n_2\mathtt{p}=p \textbf{ and } (r,v_1,v_2)=H[n_2] \textbf{ in } v_2 \\
F_a(\mathcal{S},\delta,n) & \triangleq & (n\geq 0)\ ?\ n\ :|\mathcal{S}|-1+\delta+n \\
F_d(\mathcal{H},\{\}) & \triangleq & \mathcal{H} \\
F_d(\mathcal{H},\{n,v_1,\ldots,v_k\}) & \triangleq & F_d(\mathcal{H},\{v_1,\ldots,v_k\}) \\
F_d(\mathcal{H},\{p,v_1,\ldots,v_k\}) & \triangleq & \textbf{let } (n_1,H)=\mathcal{H} \textbf{ and } n_2\mathtt{p}=p \textbf{ and } (r,v_1',v_2')=H[n_2] \textbf{ in} \\
 & & \textbf{if } \mathrm{r}=1 \textbf{ then} \\
 & & \quad F_d((n_2,H[n_2\mapsto(0,0,n_1)]),\{v_1',v_2',v_1,\ldots,v_k\}) \\
 & & \textbf{else} \\
 & & \quad F_d((n_1,H[n_2\mapsto(r-1,v_1',v_2')]),\{v_1,\ldots,v_k\}) \\
F_f(\mathrm{PC},v,n) & \triangleq & (v=0)\ ?\ \mathrm{PC}+n\ :\ \mathrm{PC}+1 \\
F_l(b_1,b_2,b_3,b_4,n) & \triangleq & F_r(((b_1*256+b_2)*256+b_3)*256+b_4)/2^n \\
F_n(\mathcal{H}) & \triangleq & \textbf{let } (n,H)=\mathcal{H} \textbf{ in } n \\
F_p(\mathcal{H},v_1,v_2) & \triangleq & \textbf{let } (n,H)=\mathcal{H} \textbf{ and } (0,0,n')=H[n] \textbf{ in } (n',H[n\mapsto(0,v_1,v_2)]) \\
F_r(n) & \triangleq & ((n+2^{30}) \bmod 2^{31})-2^{30} \\
F_s(\mathcal{S},\delta,n) & \triangleq & \mathcal{S}[F_a(\mathcal{S},\delta,n)] \\
F_t(\mathrm{PC},v,n) & \triangleq & (v\neq 0)\ ?\ \mathrm{PC}+n\ :\ \mathrm{PC}+1 \\
F_u(\mathcal{S},\delta,n,v) & \triangleq & \mathcal{S}[F_a(\mathcal{S},\delta,n)\mapsto v] \\
\mathcal{H}+n & \triangleq & \mathcal{H} \\
\mathcal{H}+p & \triangleq & \textbf{let } (n_1,H)=\mathcal{H} \textbf{ and } n_2\mathtt{p}=p \textbf{ and } (r,v_1,v_2)=H[n_2] \textbf{ in } (n_1,H[n_2\mapsto(r+1,v_1,v_2)]) \\
\mathcal{H}-v & \triangleq & F_d(\mathcal{H},\{v\})
\end{array}
$$

Figure 7: Definition of the helper functions

$$
\begin{array}{rcl}
T_b(n) & \triangleq & 0\leq n\leq 255 \\
T_c(n) & \triangleq & 0\leq n<N_c \\
T_h(\mathcal{H}) & \triangleq & F_n(\mathcal{H})<N_h \\
T_s(\mathcal{S},\delta,n) & \triangleq & 0\leq F_a(\mathcal{S},\delta,n)<|\mathcal{S}| \\
T_z(n) & \triangleq & n\neq 0
\end{array}
$$

Figure 8: Definition of the safety check functions

is presented in Figure 6. The definitions of the test functions appear in Figure 8 and those of the helper functions, in Figure 7.

Figure 6 encodes a lot of information and deserves some explanation. The semantics describe the transformation of the current state of the VEP into a new state. The current state is made of PC, $\mathcal{S}$, and $\mathcal{H}$. ($\mathcal{C}$ never changes, so it is not really a part of the state.) Only S of the current state is presented in the table. On the other hand, all of the new program counter, the new stack, and the new heap are presented. Moreover, the output produced by each expression is given in the column labeled **O**. The two remaining columns, **Saf. check** and **where . . .** , indicate particular safety checks and intermediate computations for the instructions, respectively.

Most of the instruction-wise safety is described implicitly in the table. First, there has to be a check for stack underflow (**U**) when an instruction gets one or two operands from the stack. The shape of the stack in column $\mathcal{S}$ **before** shows the operands that are consumed by each instruction. Note that an instruction requires a stack underflow check (see Figure 5) if and only if it consumes one or two operands. Second, there has to be a type check (**T**) when an instruction requires operands of specific types. The need for specific types is indicated by variables like n or p. Note that EQU and NEQ need not check the type of their arguments because we allow values of either types to be compared. Third, a check of the range of the numerical value of an operand (**R**) is explicitly indicated. DIV and MOD have to avoid divisions by zero and OUTPUT is allowed to print out bytes only. Fourth, checks for a valid access in the code space (**C**) are indicated explicitly. Instructions LOAD1 to LOAD4 need to retrieve 1 to 4 extra bytes that are part of the instruction's encoding. These instructions are the only ones that are encoded in more than one byte. Note that the branching instructions, like JUMP, do not check where they branch. This is because the new PC will be checked prior to the execution of the next instruction. Fifth, a check for a valid access in the stack space (**S**) has to be made each time a read or write access happens in the stack. Note that such R/W accesses exclude the customary consumption of the operands and production of the results ("pushes" and "pops") performed by almost all the instructions. The check consists in verifying the validity of the index where the access is about to happen. Such a check is rather complicated because an access may be performed from either the bottom or the top of the stack. So the check starts by converting the indexing operand into an *absolute* index. Sixth, a stack overflow check (**O**) is indicated implicitly only. Such a check has to be performed for instructions where the stack grows; i.e. the stack after is larger than the stack before. Seventh,

a heap overflow check (**H**) is indicated explicitly and must be performed when trying to allocate a new pair. The check verifies whether the index of the leading pair in the chain of free pairs points in H.

When all the safety checks are passed (uniform and/or specific), the instruction is allowed to have an effect on the state (and on the outside world in the case of OUTPUT). For most instructions, the effect on the state is clear except for the modifications on the heap. Here are just a few notes. The immediate constants manipulated by LOAD2, LOAD3, and LOAD4 are stored in $\mathcal{C}$ in big-endian format. In the case of PEEK, POKE, and their specialized variants, an extra adjustment δ has to be used in order to compute the absolute index where the access is to be performed. This is due to the fact that, for instance, PEEKI -3 is supposed to act exactly like LOADI -3 followed by PEEK. No -3 constant ever gets pushed on the stack but its phantom presence has to be taken into account. All arithmetic and bit-wise operations are based on 31-bit numbers that wrap around when over- or underflowing. The BSHIFT operation, when shifting to the right, works like an *arithmetic* shift; i.e. the sign bit gets extended. In the description of CONS and other operations, there is a conversion from a number, say n, to a reference to a pair, $n\mathsf{p}$, simply by adding the "p" suffix. The reverse conversion is made by stripping the suffix. Testing the type of a value v, as in ISPAIR, may simply consist in a comparison with a pair value (or a numeric value). At this point, the only difficult issue that remains is the update of the reference counts. This is performed using the mysterious $\mathcal{H}+v$ and $\mathcal{H}-v$ expressions. These expressions indicate that there now exists an extra reference to v or that we lost a reference to v, respectively. Gaining or losing a reference to a number has no impact. No update is attempted when we know that we are dealing with a number. When we deal with a pair, or when we might, the update expressions have to be used. The implementation of the these operations is rather complicated but their meaning is simple. Let us recall that, as is customary in reference counting, losing a single reference might cause a pair to be abandoned, which in turn might cause other pairs to be abandoned, and so on. In general, the loss of a reference might have a cascading effect.

8. THE VEP VERSUS OTHER VMS

There are many systems that execute untrusted codes in virtual machines to limit their access to system resources. Therefore, a question one could ask is "why not use another existing virtual machine instead of the VEP?". Here, we highlight the main reasons of choosing the VEP over two best-known virtual machines. These two virtual machines are: Java virtual machine (JVM) [6] introduced in 1995 by Sun, and the .NET platform (CLR) [7] developed more recently by Microsoft.

Any virtual machine that we choose would be a part of the TCB in EPCC framework. Knowing that any bug in the TCB can compromise the security of the whole system, we should choose a virtual machine which increases the size of the TCB the least. Using either JVM or .NET results in a large TCB (these large TCBs were the motivations for introducing the PCC approach in the first place). Appel *et al.* [2] measured the TCBs of various Java virtual machines at between 50,000 and 200,000 lines of code. The TCB size in these JMVs is even bigger than the TCB size of the traditional PCC. Therefore, using these virtual machines to extend the PCC framework results in an undesirably huge TCB and ineffective PCC framework.

For EPCC, we need a virtual machine so simple that, it is feasible for a human to inspect and verify it. None of the mentioned virtual machines and any other that we are aware of have been developed with this goal. JVM, .NET, and other well-known virtual machines are mostly focused on the performance, portability, etc. The implementation of the VEP is less than 300 lines of code which makes it possible to be easily verified by human and gives it the potential of being proven safe in future. Therefore, we have shown that the VEP is orders of magnitude smaller and it is simpler than popular virtual machines.

9. EXPERIMENT

In order to test the practicality of our approach, we have constructed a prototype implementation of an EPCC framework which uses the VEP. We needed to implement the *proof generator builder* which outputs a VEP executable proof generator. Since it would be tedious to write programs in the VEP machine language, we implemented an assembler for the VEP that allows a programmer to use instruction mnemonics instead of opcodes. Writing the proof generator program in a high-level language is even more convenient. Thus, we implemented an *assembler* to generate the machine code for the VEP, a *C compiler* which generates the assembly code for the assembler, and a *proof generator program* in the C language.

The safety proofs in PCC are represented in the Edinburgh Logical Framework (LF). The typical LF representation of the proofs is large, due to a significant amount of redundancy. The fact that proofs contain many repeated patterns of proof rules and redundant arguments, makes them suitable for data compression. A compressed proof can get decompressed using the corresponding decompressor. Therefore, a bundle of the compressed proof and a VEP machine executable decompressor which can decompress the compressed proof can make a sample proof generator. In our experiment, we used Gzip, an off-the-shelf compressor. The Gzip source code is stripped down so that it only performs the decompression task. That is, we extracted the decompressor part of the Gzip program which is called GUNzip. This GUNzip C code is given to the compiler to generate the VEP assembly code of the GUNzip. This assembly code is then given to the assembler as input which results in having the GUNzip machine code as its output. Meanwhile, the proof is compressed by the Gzip compressor. Finally, the GUNzip machine code and the compressed proof are packed together as a proof generator. The packing is done manually by allocating the compressed stream statically in the code space.

In our experiment, the proof generator machine code without the compressed data is 10KB and the proof generator bundled together with the compressed proofs average 5% the original proofs which is about 20 times smaller than before.

10. FUTURE WORK

In the future, a first practical step will be to prove the VEP safe in a PCC framework. In this way, the VEP would not increase the size of the TCB at all. Writing an oracle-based proof generator could be another possible direction to

explore. This proof generator could be one which use the proof witness in order to rebuild the original proof. Therefore, there would be no need to use any non-deterministic proof checker on the consumer side and the verification could be done with the original PCC proof checker. In this way, we would not force the consumer to change the PCC structure to gain the benefit of small proofs in OPCC and there will be no need for compromises in the size of the TCB.

11. CONCLUSION

The VEP enables the EPCC framework to make the PCC idea more scalable and practical by providing the code consumer with the luxury of using a safe environment in which a big class of proof generators can be executed in a secure manner, regardless of the original logic in which the proofs were represented. In this way, EPCC leaves the easier tasks to the consumer (like PCC) and gives adequate means to the producer to do the hard task (to the contrary of PCC). This major flexibility for the consumer and producer, in addition to the alleviation of the proof size issue, are gained through the VEP –a minor TCB extension of less than 300 lines of code– which can be easily verified and has the potential to be proved safe in a PCC framework. EPCC asks you to believe very little and gives the highest priority to the security.

12. REFERENCES

[1] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *CCS '99: Proceedings of the 6th ACM conference on Computer and communications security*, pages 52–62, New York, NY, USA, 1999. ACM.

[2] A. W. Appel and D. C. Wang. JVM TCB: Measurements of the trusted computing base of Java virtual machines. Technical Report Technical Report CS-TR-647-02, Princeton University, 2002.

[3] J. Cheney. First-order term compression: Techniques and applications, 1998.

[4] B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron. The Case for Virtual Register Machines. In *Interpreters, Virtual Machines and Emulators (IVME '03)*, pages 41–49, 2003.

[5] D. A. P. John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition, May 15, 2002.

[6] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, second edition, 2000.

[7] E. Meijer and J. Gough. Technical Overview of the Common Language Runtime, 2000.

[8] G. C. Necula. A Scalable Architecture for Proof-Carrying Code. In *FLOPS*, pages 21–39, 2001.

[9] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation*, pages 229–243, New York, NY, USA, 1996. ACM.

[10] G. C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 142–154, New York, NY, USA, 2001. ACM.

[11] H. Pirzadeh and D. Dubé. Encoding the program correctness proofs as programs in PCC technology. To appear in *PST '08: Sixth Annual Conference on Privacy, Security and Trust*, Fredericton, NB, Canada, 2008.

[12] S. P. Rahul and G. C. Necula. Proof Optimization Using Lemma Extraction. Technical report, Berkeley, CA, USA, 2001.

[13] D. Wu, A. W. Appel, and A. Stump. Foundational proof checkers with small witnesses. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 264–274, New York, NY, USA, 2003. ACM.

APPENDIX

A. SECURITY ENFORCEMENT PROCESS

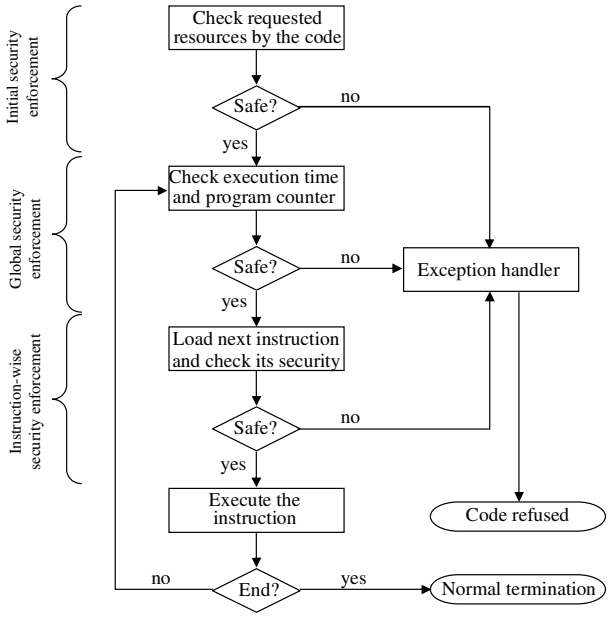

Figure 9: Security Enforcement of the VEP