An Extended Proof-Carrying Code Framework for Security Enforcement

Heidar Pirzadeh[1], Danny Dubé[2], and Abdelwahab Hamou-Lhadj[1]

[1] Department of Electrical and Computer Engineering
Concordia University
Montreal, QC, Canada
{s_pirzad, abdelw}@ece.concordia.ca
[2] Department of Computer Science
Laval University
Quebec City, QC, Canada
danny.dube@ift.ulaval.ca

Abstract. The rapid growth of the Internet has resulted in increased attention to security to protect users from being victims of security threats. In this paper, we focus on security mechanisms that are based on Proof-Carrying Code (PCC) techniques. In a PCC system, a code producer sends a code along with its safety proof to the consumer. The consumer executes the code only if the proof is valid. Although PCC has been shown to be a useful security framework, it suffers from the sheer size of typical proofs -proofs of even small programs can be considerably large. In this paper, we propose an extended PCC framework (EPCC) in which, instead of the proof, a proof generator for the program in question is transmitted. This framework enables the execution of the proof generator and the recovery of the proof on the consumer's side in a secure manner using a newly created virtual machine called the VEP (Virtual Machine for Extended PCC).

Keywords: Software Security, Proof-Carrying Code, Virtual Machine.

1 Introduction

Modern computer systems have become so complex that traditional security mechanisms built around anti-viruses and intrusion detection mechanisms can no longer sustain the severity of today's ever-increasing security threats. One can claim that, except perhaps for security experts and professionals, it is too big a burden, or even unrealistic, for users to bear sole responsibility for adequate security and protection of their computing systems. Proof-Carrying Code (PCC) techniques have been introduced to reduce the impact of this problem by allowing a consumer of a computer program to verify a proof of its general safety properties, sent by the code producer, before executing it [8].

In a PCC system, there are typically two main parties, (1) a code producer, who builds machine code along with its safety proof (expressed typically in a formal

M.L. Gavrilova et al. (Eds.): Trans. on Comput. Sci. XI, LNCS 6480, pp. 249–269, 2010.
© Springer-Verlag Berlin Heidelberg 2010

logic), and (2) a code consumer, who wishes to run the compiled code as long as it satisfies predetermined safety policies.

A typical interaction between the producer and consumer encompasses several steps. In the first step, the producer sends the consumer a program, which consists of the code and additional annotations such as loop invariants and function pre- and post-conditions. The consumer provides the received code to the Verification Condition Generator (VCGen), which generates a verification condition based on a set of safety policies that need to be satisfied. A verification condition is a logical formula that, if satisfied, implies that the code satisfies the safety policies.

The consumer, then, sends the generated verification condition to the producer. The producer runs a theorem prover (in many cases along with necessary human intervention) to obtain a proof that corresponds to the received verification condition. Next, the producer submits the proof to the consumer. The consumer uses a proof checker to verify that the received proof is indeed a proof of the verification condition that was initially generated. If the check succeeds the code is considered trustworthy and can be executed.

It should be noted that it is very common to have a copy of the VCGen on the producer's side to simplify the interaction between the code producer and the code consumer. In this way, the code consumer receives the annotated code as well as the safety proof during the first step of the interaction. Fig. 1 shows by the components according to the order by which they are executed in the PCC process. The steps involved in a typical interaction between producer and consumer as discussed above. The ovals are the artifacts that are generated/sent, the arrows represent the flow of the artifacts, and the rectangles show the components that perform computations. The starting point of the interaction is represented by a closed circle (•). At the end of the interaction, a switch (symbolically shown as a triangular tri-state buffer) checks the result of the proof checking; if the proof checking succeeds the code is considered trustworthy and can be executed (on the CPU shown as a rhomboid) if not the switch remains off and the code will not be executed.

One of the key properties of a PCC framework is that the Trusted Computing Base (TCB) (specified by the orange curved rectangle in Fig. 1) contains relatively small and simple components such as VCGen and a proof checker while the theorem prover is on the producer's side and therefore out of the consumer's TCB. The reason for that is twofold: performance and security. That is, in general, proving the verification condition is a resource consuming task which can result in low performance. Furthermore, considering that the theorem prover is a large and complex program, it could not be placed on the consumer's side as it could hardly be trusted. Another important property of the PCC framework is that PCC programs are tamper-proof. An intruder cannot modify the code or the proof in a way that results in execution of a malicious code on the consumer's side. Any attempt to tamper with either the code or the proof results in a validation error during the proof checking process.

Despite the fact that PCC can be a powerful security mechanism, it is still not widely accepted in practice due to two keys issues. First, it is usually difficult to write proofs for large programs. Although with the recent advances in Certifying compilation [3, 28] some safety properties of programs can automatically be proved as certificates, this is limited to basic safety properties and only possible for a restricted class of programs. For example, it may not be possible to prove automatically safety properties if the software system is complex or the policies are sophisticated [29]. The

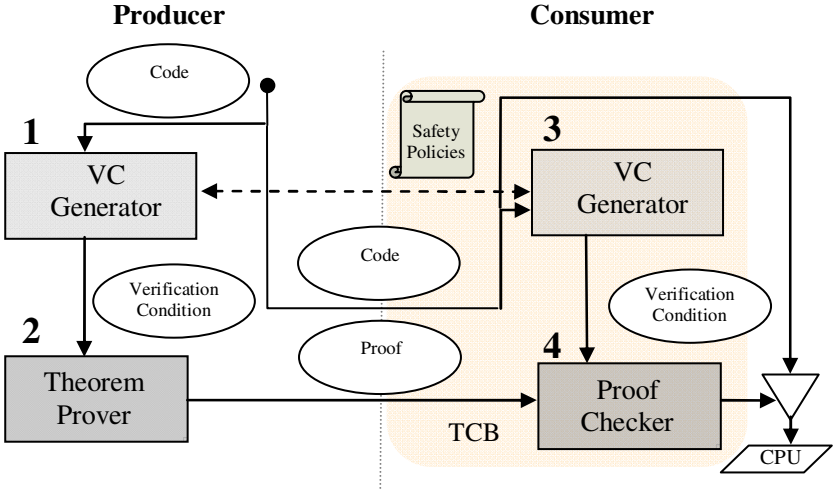

Fig. 1. Conventional PCC framework: typical steps and involved components

second limitation, which is the topic of this paper, is concerned with the difficulty in communicating and storing the proofs which are inherently large [11]. It is common to have proofs that are 1000 times larger than the associated code, which renders the use of PCC impractical for all but the tiniest examples [11]. This is further complicated when dealing with systems with limited storage and processing resources such as mobile and handheld devices, and networks with low survivability and scarce resources.

Clearly, there is a need for efficient techniques to reduce the size of proofs. The approaches proposed to alleviate this issue which include the use of data compression techniques [8, 10, 11, 25] suffer from drawbacks of their own, among which the most important one is the enlargement of the TCB, A large TCB increases the chance of defects which may cause an unsafe program to be accepted.

In this paper, we propose a novel approach to solving the proof size problem while avoiding to increase significantly the TCB. Our approach is based on the innovative idea of sending a program that generates the proof instead of the proof itself. This is inspired by the concept of Kolmogorov complexity [16], where the complexity of a string x is the shortest computer program that produces x on a so-called universal computer, i.e., a machine that computes the string, prints it, and then halts. One important observation is that the ideal compressed form for a given proof is the shortest program that outputs that proof.

To allow the proof generator program to execute on the consumer's side, we have developed a virtual machine that we call VEP (the Virtual Machine for Extended PCC). VEP is written in C and has less than 300 lines of code, which is an acceptable addition to the consumer's TCB. The design of VEP is relatively simple to be able to easily verify that is safe. It has also been developed with security in mind so as the running programs do not access unauthorized resources. Using the VEP, we believe that proofs, which are represented as programs, can be executed safely on the consumer's side while keeping the consumer's TCB reasonably small.

Organization of the paper: In the next section, we provide background information about PCC, and discuss studies related to our work. In Section 3, we present the extended PCC framework, followed by the VEP and its components. We show the effectiveness of our approach by applying it to several benchmark proofs in Section 4. We conclude the paper and discuss future directions in Section 5.

2 Background and Related Work

It is desirable that proofs be represented in a compact format. One way to reach this goal is through proof optimization in which the proofs are rewritten in a more compact form which preserves the meaning of the original form of the proof [13, 2]. This could be done by replacing all the occurrences of a given term t with a smaller equivalent term 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). Necula et *al.* experimented with proof optimization in an approach called lemma extraction and were able to obtain a minor reduction gain of 15% in the size of the proofs [2].

Another way of compacting the proofs is through data compression. Data compression techniques 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, which requires a safe decompressor on the consumer's side. Generating the proof of safety for a normal decompressor (a relatively large program with about 7000 lines of code) can be a difficult task not worth performing because one would only obtain a specific decompressor that cannot work with a proof compressed by an appropriate but different compressor. In other words, each time a new decompressor is used, a proof of its safety is required. The objective of the VEP is to tackle this problem by tailoring it to the needs of executing proof generators that could be, as shown in our case study, a compressed file along with a decompression tool.

Necula et *al.* proposed a new strategy called Oracle-based Proof-Carrying Code (OPCC) [11]. In OPCC, the handling of the proofs on the consumer's side is changed. As shown in Fig. 2, this change in strategy, led to a change in the framework, namely, they assumed that the consumer uses a non-deterministic proof checker.

The untrusted theorem prover on the producer's side records a sequence of bits that shows which sub-goals failed and needed backtracking. Then, the producer sends to the consumer this bit-stream that serves as a proof witness. On the consumer's side, the received bit stream works as an "oracle" which can guide the trusted non-deterministic proof checker to avoid back-tracking. Every time the checker must make a choice between the possible ways to proceed, it consults some bits from the oracle. In this approach, the trusted non-deterministic proof checker is, in fact, a non-deterministic theorem prover having the task of proving the verification condition. The oracle is used to drive the theorem prover to a final proof without search.

Experimental evidence shows that oracle strings, as suggested by Necula et *al.*, can be about 1/8 of the code size and about 30 times smaller than proofs in traditional PCC [11]. However, Wu et *al.* [14] suggested that the code size relation might be deceptive as the size Java class files, that are necessary to be sent along with the proof witness in a SpecialJ proof-carrying Java system, is not included in calculation.

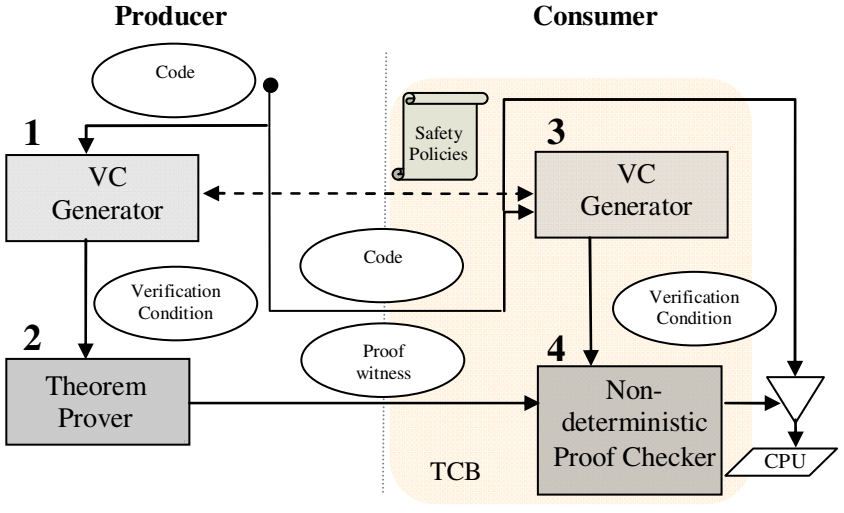

Fig. 2. OPCC framework: typical steps and involved components

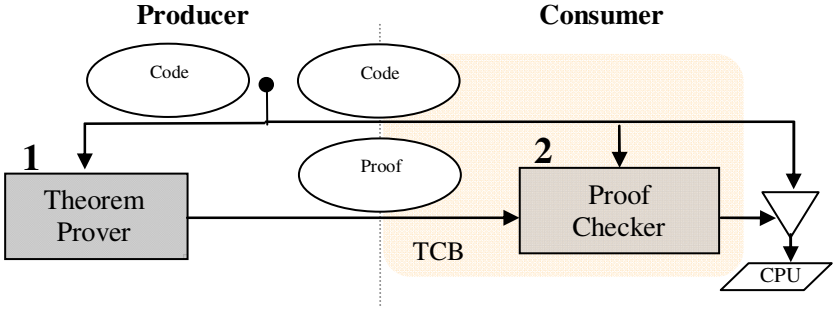

Fig. 3. FPCC framework: typical steps and involved components

One of the most important downsides of the OPCC is that it involves complex trusted components, such as a non-deterministic proof checker plus the usual PCC components. The TCB in OPCC is about 26000 lines of C code which is larger than the TCB size in traditional PCC (15000-20000 LOC). Any flaw in the implementation of these components can compromise safety of the system. As a matter of fact, the Special-J system [3], used in Necula et *al.*'s approach, showed a critical leak in its type axioms found by League [5].

Although the above approach has resulted in proofs which were smaller than the original proofs, they had to significantly enlarge the TCB. In fact, Appel points out that the VCGen (and consequently the TCB size) even in traditional PCC is too large [27] and it needs to be verified. As shown on Fig. 3, Foundational Proof-Carrying (FPCC) [27] Code aims to further reduce the TCB size by removing the VCGen from the consumer's side.

FPCC uses a foundational mathematical logic for defining the semantics of the machine instructions and the proof rules. In this way, Appel et al. avoid using the VCGen by defining the operational semantics of machine instructions and the safety policies in a higher-order logic. This is done by modeling the machine instruction with a transition from one machine state (set of memory and registers) to another machine state and defining the safety policy accordingly. Similar to the PCC, a theorem prover should produce a proof of safety to be accompanied by the code. The proof checker verifies the safety proof before the program is executed. FPCC is concerned with minimizing the TCB of the system, by not including the VCGen as shown in Fig. 3.

While the original FPCC uses deductive reasoning to encode proof rules, some variants of FPCC use computational reflection to replace deduction by computation [31]. FPCC is likely to be more secure than traditional PCC because it has a smaller TCB. However, the proofs in FPCC, in comparison with traditional PCC, are more complex to produce and, as stated by Appel et *al.*, can explode exponentially [27]. According to Necula, the proof size in FPCC is 20% bigger than the proof size in traditional PCC [11]. Therefore, even though in FPCC, it is only necessary to send a proof generator, the complexity of producing the proofs, in the first place, renders FPCC hard to use in practice.

Wu et *al.* [14] proposed submitting annotated programs that can be checked for safety by a verified logic program. The program logic clauses are derived as lemmas from the (trusted) axioms about safe execution of the machine. This way, it is not necessary to build and check a large proof at the code consumer's side. However, according to [32], there exist issues about scalability of the results, as reported by Wu et *al.* [14], and effective engineering of their verifiers.

While we are not in favor of possible compromises to the security of the system due to a large TCB expansion (as we have in OPCC), we also like to overcome the difficulty in communicating and storing the proofs which are inherently large (as we have in traditional PCC and more severely in conventional FPCC) in a practical way.

3 The Extended Proof-Carrying Code Framework (EPCC)

3.1 Overview

Fig. 4 describes the steps involved in the proposed Extended Proof-Carrying Code (EPCC) framework [17]. In an EPCC system, there are two main parties, a code producer, on the left-hand side, who sends a code along with its safety proof generator program[1], and a consumer, on the right-hand side, who wishes to run the code provided that it is proven safe by the system.

The interaction between these two parties consists of the following steps. In the first step, the producer runs a theorem prover to obtain a safety proof of the code he intends to send. Similar to what is done in FPCC [27], the producer is not constrained to generate the safety proof in the logic that the consumer imposes. The producer can

[1] A proof generator is a program whose sole function is to output the proof. This program aims to be a more compact representation of its resulting proof and does not necessarily rediscover the proof.

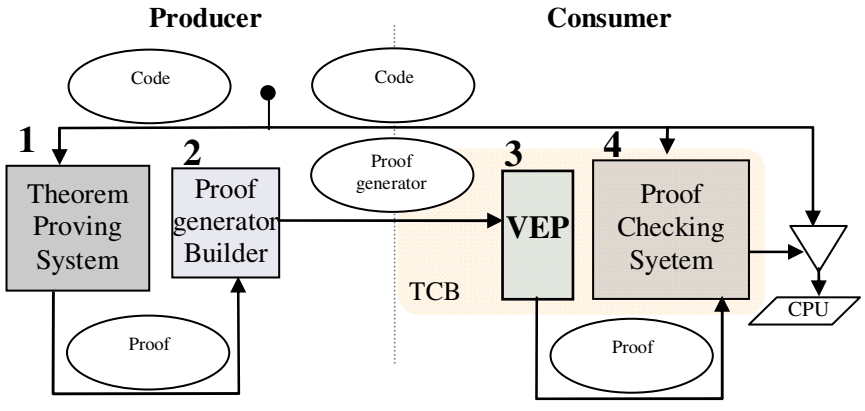

Fig. 4. EPCC framework: typical steps and involved components

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. In the second step, the producer writes a proof generator program, which outputs the safety proof in the format which is acceptable by the consumer.

Next, the producer submits the code accompanied by its safety proof generator program to the consumer. At this point, the proof generator program is yet another program that the producer sends to the consumer. It is as untrustworthy as the payload code itself. So it seems we are in a kind of chicken-and-egg situation: before running the untrustworthy payload code, the consumer needs to verify its attached proof, which requires execution of the proof generator program, which is also untrustworthy. One possible way to overcome this issue is to simply verify the safety of the proof generator program using traditional PCC. This solution has the obvious drawback of necessitating a proof for each proof generator program, which could hinder the practical aspect of our approach due to the complexity of writing proofs. We propose, instead, to run the proof generator in a tightly sandboxed environment: our carefully designed virtual machine, the VEP (the Virtual machine for Extended Proof-carrying code). The design of the VEP is discussed in more details in Section 4.

Upon receiving the code and the corresponding proof generator program, the consumer runs the proof generator (only for a single time) on the VEP and obtains the safety proof. The next steps are similar to the traditional PCC: The consumer runs the proof checker; after the proof check succeeds the consumer can safely execute the code. As one can easily observe, the EPCC framework is tamper proof, just like PCC.

The EPCC framework not only makes PCC more scalable and practical by reducing the proof size but also provides the code consumer with the possibility to use a safe environment in which a large class of proof generators that can be executed in a secure manner, regardless of the original logic in which the proofs were represented. In this way, EPCC leaves the easiest tasks to the consumer and gives adequate means to the producer to do the hard tasks. This major flexibility for the consumer and producer is gained through the VEP, a minor TCB extension, which can be verified

easily. Technically, except for the VEP, the security of EPCC is as strong as the traditional PCC. Currently, a verified VEP is being developed (using conventional PCC). A verified VEP would potentially make EPCC exactly as secure as PCC.

4 Virtual Machine for Extended Proof-Carrying Code (The VEP)

The VEP [12] is intended to be a sandbox interpreter for the proof generator programs. Any defect in the VEP might give an opportunity to an attacker to write a malicious proof generator such that its execution on the VEP turns the VEP into an attacker against the consumer. Therefore, the safe execution of the proof generator depends greatly on the safety of the VEP and the way it imposes the security requirements. In this section, we present the design of the VEP starting from the general requirements that the VEP needs to satisfy to be deemed secure.

4.1 Requirements

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

1. The VEP should run as a virtual machine, deployed on different platforms to allow portability of proofs. This is similar in principle to the concept of universal computing proposed by Kolmogorov when describing the characteristics of the ideal decompressor [16].
2. It should enable the execution of the proof generator at the consumer's side in a secure manner. It should provide a tightly controlled set of resources for proof generation. Network access, the ability to inspect the host system, or read from input devices and write into file streams should be disallowed. Moreover, the VEP should be able to perform some sort of execution monitoring to verify that these constraints are maintained.
3. As indicated in EPCC framework, the VEP is a part of the TCB of the consumer. Knowing that any bug in TCB can compromise the security of the whole system, we need the VEP to be small and simple such that it is relatively easy to check for its safety. This would give the VEP the potential to be proved safe by the PCC itself.
4. The proof generators are sent in the VEP language. Consequently, this language should be flexible enough so that it allows compact proof generators to be written.
5. The VEP should be designed with performance in mind since it adds an overhead to the processing of proofs on the consumer's side.
6. The design of the VEP should be based on proven practices and common technologies to facilitate its adoption.

It should be noted that the above requirements are not equally important. The three first requirements are the most important ones in case trade-offs need to be made. For example, the low complexity and small code size both depend on the number of instructions in the VEP instruction set. On one hand, having a small set of instructions results in a virtual machine with low complexity, on the other hand, a large list of instructions makes the code smaller. Although these two factors are contradictory,

there can be a good balance between them. Therefore, finding good trade-off has been one of the guiding principles in designing the VEP.

4.2 Machine Type

Conventionally, a virtual machine (VM) can either be stack-based or register-based. Implementing a universal computer can be achieved with a stack machine which has more than one stack or has one stack with random access. Nevertheless, register machines can be universal computers; therefore, both approaches can satisfy Requirement 1.

The most popular virtual machines, however, such as the Java Virtual Machine [6] and the Common Language Runtime [7], use a stack machine type rather than the register-oriented architectures due to the simplicity of their implementation. Hence, a stack-based machine helps us to better fulfill Requirement 3 (simplicity of the design). 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 (for execution on register machines) can be translated into an equivalent stack machine program. Moreover, the stack machines are easier to compile to, which could potentially help the adoption of the VEP (Requirement 6).

Finally, we chose the stack machine type over the register one because a compiled code for a stack machine has more density than the one for the register machine. In an experiment, Davis et *al.* [4] showed that the corresponding register format code after eliminating unnecessary instructions was around 45% larger than the stack code needed to perform the same computation. This can especially affect the size of the proof generator written for the VEP as mentioned earlier.

4.3 Instruction Set Architecture

The Instruction Set Architecture (ISA) of a virtual machine is the VM interface to the programmer. In the case of the VEP, available data types and the set of memory spaces are defined by ISA. The ISA definition also includes the 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.

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 word shows the data type of the stored value. This bit is not visible to the programmer while the remaining 31 bits are visible. If we have a word that references a *pair*, the content of the word represents the address of a *pair* in memory. For a word with its type *number*, the content of the word is a signed integer.

Memory
The VEP uses three blocks of memory: a code space (an array whose elements are bytes), a heap (an array whose elements are pairs, see below), and a stack (an array whose elements are bytes).

A *pair* is an ordered sequence of two values; the representation includes two words, for both values, and a third word which stores the reference counter for the simple garbage collection system of the VEP.

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.

Fig. 5 shows the schemata of the stack and the heap in the VEP. For each of these two schemata, sample binary contents 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*[2] 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.

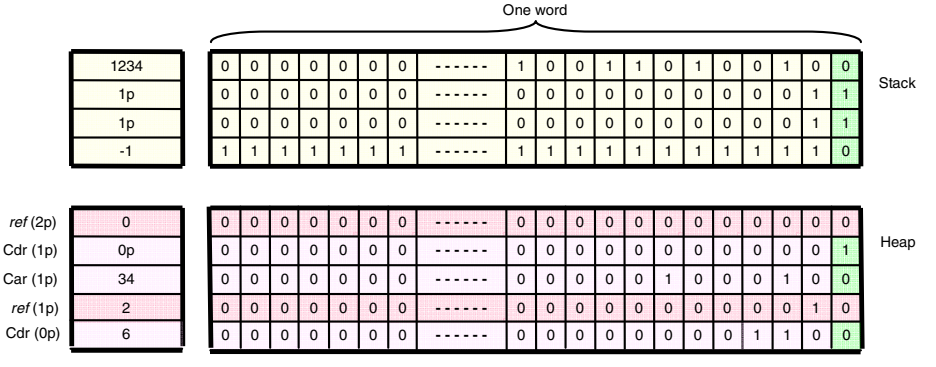

Fig. 5. Schemata of the stack and the heap

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.

The VEP relies on the reference counting [23] to automatically detect unused objects and collect them from memory. A major drawback of reference counting is its failure in reclaiming cyclic garbage data structures. We took the care of designing the VEP so that it does not have the ability to perform *destructive updates* on the pairs. 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. This way,

[2] Analogous to 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.

the weakness of the reference counting garbage collector is avoided due to the lack of circular structures in the VEP heap.

Instruction Set

The design of the instruction set is one of the most interesting and important aspects of the 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, many 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 when developing VEP-enabled proof generators. It also guarantees an acceptable execution performance (that is in line with 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 distribution of the instructions is based on an interpretation of the work of Hennessy et al [18] where they found the 10 simple instructions that account for 96% of the instructions executed for a collection of integer programs running on the popular Intel 80x86. We used Table 1 as a reasonable guide for determining an appropriate distribution of instructions (in line with requirement 6).

Table 1. Distribution of instructions interpreted from [18]

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%

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 can perform their operations only on the top eight elements of the stack), while `PEEK` and `POKE` have broader range (they can perform their operations only on all elements of the stack) 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, process instructions sequentially. 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`): These instructions compare values by using a specific comparison operation. Typical comparison instructions 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), constructs a pair (`CONS`), or verify if a value 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.

Almost all of the instructions take their arguments from the stack and have no (immediate) operands. In particular, `PEEK`, `POKE`, and jump instructions are intended to be used along with "`LOAD*` *`val`*;" instruction[3]. This keeps almost all of the instructions to a single variant (no need to handle various addressing modes). Prevalence of `LOAD*` explains the existence of the 1-byte instruction `LOADI` for constants close to zero. These choices achieve simplicity of the VEP and compactness of the byte-code. The only instructions with immediate operands (other than `LOAD*`) are `POKEI` and `PEEKI`, which are extremely frequent as they are the typical means to implement the write/read of the local variables on the stack.

A note-worthy point about the VEP instructions set is the absence of instructions which operate on network or gives the ability to inspect the host system. Furthermore, there are no instructions which can read from input devices and write into file streams. These are to enable the execution of the proof generator at the consumer side in a secure manner. That is, we tried to enforce security policies such as no access to files or no access to the network on instruction set design level. Thus, the selected instructions provide the VEP with a tightly-controlled environment for proof generator to run in.

4.4 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

[3] `LOAD*` *`val`* pushes the numeric value encoded by the next * byte(s) in the code space onto the stack.

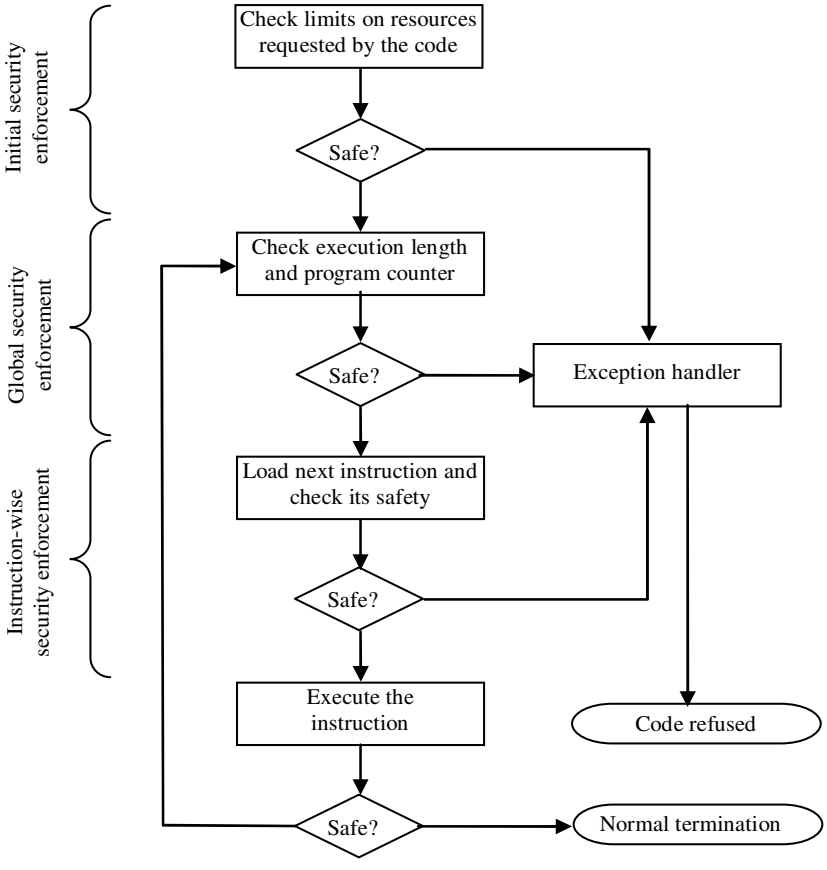

Fig. 6. The flowchart of Security Enforcement of the VEP

0, . . . , $Nc - 1$, where Nc 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 is a valid destination varies from instruction to instruction. Generally, the valid read and write destinations are stack locations ranging from the bottom 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.

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. Fig. 6 shows a complete schema of the security enforcement mechanism and its different levels.

Initial Security Enforcement

A proof generator makes requests for resources. These requests are made using a declaration in the header of the proof generator. Each time, the VEP verifies whether the requested amount of resources is no greater than the maximum value settled in an agreement between the producer and the consumer. The requested code size and stack size are, respectively, denoted by *Nc* and *Ns*. The amount of needed heap size of the proof generator is represented in number of pairs *Nh*.

- *Code size*: If the VEP refuses or fails to allocate the requested block of memory, the VEP refuses the proof generator. Otherwise, the VEP allocates a block of *Nc* 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 proof generator. Otherwise, the VEP allocates a block of *Ns* 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 generator. Otherwise, the VEP allocates a block of 3**Nh* words of memory as the heap memory.
- *Length of Execution*: The proof generator should finish its task within a definite number of operations *No*. In the case where the *No* is more than the limit the VEP refuses the proof generator.

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

Global Security Enforcement

Throughout the execution, the VEP enforces two security checks globally, which are independent of the next instruction that is about to be executed. The global security enforcement consists of checking the following aspects:

- *Length of execution*: Before fetching the next instruction, the VEP makes sure that the elapsed time of the execution of the proof generator (measured as the number of executed operations) has not exceeded the number of operations (i.e., *No*). 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).

Instruction-Wise Security Enforcement

The third level of security enforcement by the VEP is the fine-grained level and is done per instruction. This level of security prevents the proof generator from performing 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 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 proof generator is not considered safe.
- *Type of operands*: The VEP checks if the type of the operands conforms to 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 free of type-mismatches according to the VEP's type system.
- *Legal range of operands*: The arithmetic instructions should have legal arguments. The VEP checks the operand legality to prevent potential errors 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 a 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 space 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.

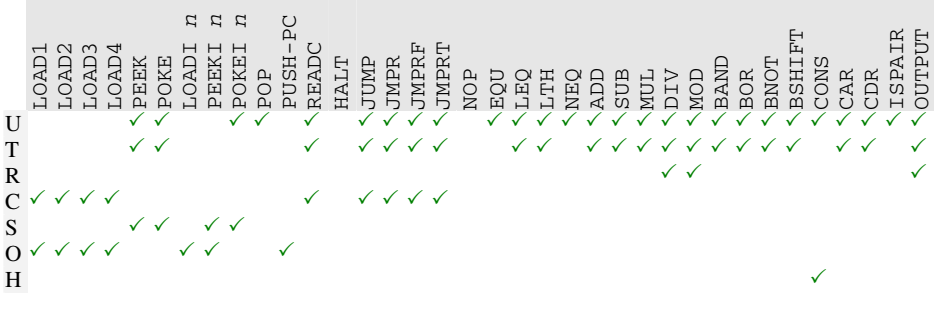

Fig. 7. Instruction-wise security enforcement

As shown in Fig. 7, the complete set of instructions[4] 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.

4.5 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 popular virtual machines, which are the Java virtual machine (JVM) [6] and the .NET platform (CLR) [7].

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.* [1] measured the TCBs of various Java virtual machines at between 50,000 and 200,000 lines of code. The TCB size in these VMs is even larger than the TCB size of the traditional PCC. Therefore, using these virtual machines to extend the PCC framework would result in an undesirably large TCB and hence an 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 or any other ones that we are aware of has been developed with this goal in mind. JVM, .NET, and other well-known virtual machines focus essentially on performance, portability, etc. Similar to other components of the TCB in traditional PCC and OPCC, the VEP is implemented in C language. However, unlike the OPCC that extends the TCB for about 9000 lines of C code, the implementation of the VEP is less than 300 lines of code which makes it possible to be easily verifiable by humans and gives it the potential of being proven safe in the future. Therefore, we have shown that the VEP is orders of magnitude smaller and it is simpler than popular virtual machines.

5 Application of EPCC

The proofs in PCC are commonly represented in the Twelf format [26] (an implementation of the Edinburgh Logical Framework (LF) [36]). We applied our approach to six proofs (see Table 2) produced by a solver made available by Aaron Stump[5]. The solver accepts quantified Boolean formulas benchmarks in the standard QDIMACS format, and emits proof terms showing whether the formula evaluates to true or to false. These proofs are the same as the ones considered in Stump's work [15], where easy benchmark formulas from [21] different domains (formal verification, planning, etc)[6] were solved to generate the proof terms. All proofs use a form of implicit LF [9]

[4] All 256 available opcodes are assigned to these 36 instructions; few instructions with immediate argument cover more than one opcode as the argument is encoded in the opcode itself.

[5] http://www.cs.uiowa.edu/~astump/software.html

[6] Interested readers can see [30] for a complete description of the domains and families of the proved formulae.

and can be as large as 7.4 megabytes. Although these proofs were not specifically designed for PCC, we believe that they can be fair representatives of large proofs, and be used in the absence of large PCC proofs due to the complexity of building them.

5.1 Building a Proof Generator

We created a proof generator for each of Stump's proofs. Our proof generator consists of a package that comprises a compressed version of the proof and a VEP machine executable decompressor. That is, we built a self-decompressing executable program which will generate the original proof as a result of being executed on the VEP. For this purpose, we reused an existing off the shelf compression tool, Gzip [22], which we modified to make it VEP-enabled. We could have also created our own program that generates the proof by looking at patterns in the proofs and creating programs that would explore these patterns forming a compact representation of a proof as a running program. We deliberately chose not to proceed this way to show that our framework can be equally used with existing programs, relieving the users of our framework from creating proof generators from scratch. However, we recognize that one of the main drawbacks of our approach lies in the need to adapt any program used to represent a proof to the VEP, a task that may turn to be difficult and time consuming. There is definitely a need to further investigate this issue as a key future direction.

Fig. 8 shows the steps involved in EPCC. The first and second steps are similar to traditional FCC. Given a proof, in the 3rd step a component called "proof generator builder" indicated by as a box with upward diagonal pattern in is responsible for building a proof generator. As shown in Fig. 8, the proofs are compressed using Gzip. To decompress the proofs on the consumer's side, we needed to send the decompression tool that can run on the VEP along the compressed proofs. For this purpose, we modified Gzip component that performs the decompression task (called gunzip). This involved using static allocation, removing all preprocessor commands and function prototypes, in-lining functions, etc. In order to in-line the functions without causing an increase in the code size, we used the `computed goto` construct [24], which is a `goto` statement for which the address of the target is computed by an expression of type `void*`.

The modified decompressor fetches its input (compressed data) from a literal string (array of compressed data) and outputs the decompressed data on the standard output. For the decompressor to fetch its input from a literal string, and to print a character, respectively, `readcmp` and `putchar` were developed as two special functions.

The modified gunzip C code (which now contains about 2000 LOC) is re-compiled to generate the assembly code of the gunzip (see Fig. 8). For this, we developed our own C compiler that supports a subset of C constructs that map to the VEP instruction set. The C compiler is based on the C89 open source complier [20]. Since the computed `goto` is not supported by the ANSI C89 grammar, we added it to the C89 grammar.

The assembly code generated by the compiler is then given to the assembler as input which results in having the VEP-executable gunzip machine code as its output (see Fig. 8). Our assembler implemented in C, permits *assembly-time* arithmetic operations to take place in order to compute constants to include in the assembled

program. Thus, the expressions are evaluated during the assembly and the results become permanent parts of the code.

Gunzip machine code and the compressed proof are packaged to form a proof generator sent to the consumer. This packaging is performed manually by allocating the compressed stream statically in the code space. This saves us a lot on stack space in comparison with the case we dynamically allocation the compressed data in a global variable. The compressed stream is then read by the decompressor using the auxiliary function `readcmp`. This is the only function that we add to the existing decompressor code so that it can read the compressed data from within the decompression code.

Before sending the proof generator, the producer needs to add the request in code size, heap size, stack size, and execution time to the proof generator program header. For this, he has the option of running the proof generator on a copy of the VEP installed on his side. The VEP contains a feature that can add automatically the actual amount of the consumed resources to the proof generator program header.

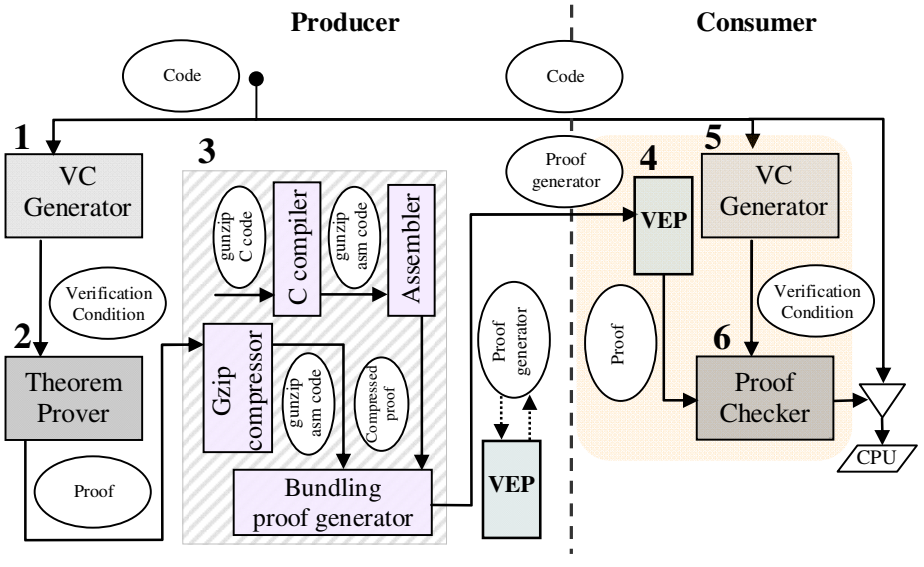

Fig. 8. Detailed diagram of our sample implementation of EPCC

5.2 Results of Applying the Approach

Table 2 shows the results of applying our approach to the proofs selected for this study. For each proof, the original proof size (N) and the size of the proof generator (NPG) are represented.

The size of the proof generator excluding a compressed proof is about 15KB (which is the size of gunzip machine code and is constant for all of our proof generators). The proof generators average 2.9% the original proofs which is about 34 times smaller than before, which constitutes a significant gain in size reduction. The proof generator reduction in size relative to the original size of the proof is represented as the percentage of space savings (*SS*):

$$Space\ Savings = 1 - (NPG / N)$$

The space saving ratio of proof generators to the size of the original proofs ranges from 87.19% up to 96.77%. The table also shows the elapsed times of the execution of proof generators on the VEP. All times are reported in seconds on an Intel Core Duo CPU 2.00GHz, 2MB cache, 1GB main memory, running Windows XP. We can see that the VEP performed in less than a second for processing the proof generators.

Table 2. The size effect of representing proofs as programs

Experiment	N	NPG	SS %	Elapsed time	Domain
cnt01e	164 KB	21 KB	87.19	< 1s	Formal verification
tree-exa2-10	337 KB	25 KB	92.58	< 1s	Pattern matching
toilet 02 01.2	917 KB	45 KB	95.09	< 1s	Planning
1qbf-160cl.0	1407 KB	59 KB	95.80	< 1s	Formal verification
tree-exa2-15	3847 KB	115 KB	97.01	< 1s	Pattern matching
toilet 02 01.3	7377 KB	238 KB	96.77	< 1s	Planning

6 Conclusion and Future Work

In this paper, we presented an extension to a traditional proof-carrying code framework in which proofs tend to be considerably large to transmit. Our extended framework is based on the idea of representing proofs and programs that are sent to the consumer. As such, the consumer runs the program and generates the original proof. The proof generator program should be the shortest possible to maximize the size reduction gain.

We developed a virtual machine called the VEP that runs on the consumer's side and which is responsible of running the proof generator program. The implementation of the VEP contains less than 300 lines of code which is a minor extension to the consumer's TCB.

The VEP enables the proposed extended PCC framework to make the PCC idea more scalable and practical by providing the code consumer with the possibility of using a safe environment in which a large class of proof generators can be executed in a secure manner, regardless of the original logic in which the proofs were represented.

In the future, a first practical step will be to obtain a VEP that has been proven safe using the conventional 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 uses 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. When both the proof generator and the proof checker can work incrementally, the whole proof need not be rebuilt at any one time on the consumer side. Instead, the output of the proof

generator can be piped into the input of the proof checker, which consumes (and verifies) parts of the proof as soon as they are output.

In addition, we intend to continue experimenting with the proposed approach using larger proofs. This can be hard to achieve due to the unavailability of proofs for large systems.

Finally, we intend to compare the results of our approach with existing approaches such as the oracle PCC [11], although the size reduction gain should not be the only criterion that needs to be used in the comparison since, again, any approach that increases considerably the TCB poses risks to security no matter the size compression ratio achieved.

References

1. Appel, W., Wang, D. C.: JVM TCB: Measurements of the trusted computing base of Java virtual machines, Tech. Rep. CS-TR-647-02, Princeton University (2002)
2. Cheney, J. R.: First-order term compression: techniques and applications, Master's thesis, Carnegie Mellon University (August 1998)
3. Colby, C., Lee, P., Necula, G.C., Blau, F., Plesko, M., Cline, K.: A certifying compiler for Java. SIGPLAN Not. 35(5), 95–107 (2000)
4. Davis, B., Beatty, A., Casey, K., Gregg, D., Waldron, J.: The case for virtual register machines. In: Proceedings of the 2003 Workshop on interpreters, Virtual Machines and Emulators, IVME 2003. San Diego, California, June 12 - 12, pp. 41–49. ACM, New York (2003)
5. League, C., Shao, Z., Trifonov, V.: Precision in practice: a type-preserving java compiler. In: Hedin, G. (ed.) Proceedings of the 12th International Conference on Compiler Construction, Warsaw, Poland, April 07-11. Lecture Notes In Computer Science, pp. 106–120. Springer, Heidelberg (2003)
6. Lindholm, T., Yellin, F.: Java Virtual Machine Specification, 2nd edn. Addison-Wesley Longman Publishing Co., Inc. (1999)
7. Meijer, E., Gough, J.: Technical Overview of the Common Language Runtime (2000)
8. Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1997. Paris, France, January 15-17, pp. 106–119. ACM, New York (1997)
9. Necula, G.C.: A Scalable Architecture for Proof-Carrying Code. In: Kuchen, H., Ueda, K. (eds.) FLOPS 2001. LNCS, vol. 2024, pp. 21–39. Springer, Heidelberg (2001)
10. Necula, G.C., Lee, P.: Safe kernel extensions without run-time checking. SIGOPS Oper. Syst. Rev. 30, 229–243 (1996)
11. Necula, G.C., Rahul, S.P.: Oracle-based checking of untrusted software. In: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2001. London, United Kingdom, pp. 142–154. ACM, New York (2001)
12. Pirzadeh, H., Dubé, D.: VEP: a virtual machine for extended proof-carrying code. In: Proceedings of the 1st ACM Workshop on Virtual Machine Security, VMSec 2008. Alexandria, Virginia, USA, October 27-27, pp. 9–18. ACM, New York (2008)
13. Rahul, S.P., Necula, G.C.: Proof Optimization Using Lemma Extraction. Technical Report. UMI Order Number: CSD-01-1143., University of California at Berkeley (2001)
14. Wu, D., Appel, A.W., Stump, A.: Foundational proof checkers with small witnesses. In: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming, PPDP 2003. Uppsala, Sweden, August 27-29, pp. 264–274. ACM, New York (2003)

15. Stump, A.: Proof Checking Technology for Satisfiability Modulo Theories. Electron. Notes Theor. Comput. Sci. 228, 121–133 (2009)
16. Li, M., Vitnyi, P.: An Introduction to Kolmogorov Complexity and its Applications, vol. 3. Springer Publishing Company, Heidelberg (2008) (incorporated)
17. Pirzadeh, H., Dubé, D.: Encoding the Program Correctness Proofs as Programs in PCC Technology. In: Proceedings of the 2008 Sixth Annual Conference on Privacy, Security and Trust, October 01-03, pp. 121–132. PST. IEEE Computer Society, Washington (2008)
18. Hennessy, J.L., Patterson, D.A.: Computer Architecture: a Quantitative Approach, vol. 3. Morgan Kaufmann Publishers Inc., San Francisco (2003)
19. Jansen, W., Karygiannis, T.: NIST special publication 800-19 – mobile agent security. Technical report, National Institute of Standards and Technology, Computer Security Division, Gaithersburg, MD 20899. U.S. (2000)
20. American National Standards Institute, "Programming Language C," Document ANSI X3.159-1989
21. Giunchiglia, E., Narizzano, M., Tacchella, A.: Quantified boolean formulas satisfiability library (qbflib) (2001), `http://www.qbflib.org`
22. Deutsch, P.: GZIP File Format Specification Version 4.3. RFC. RFC Editor (1996)
23. Christopher, T.W.: Reference count garbage collection. Software – Practice and Experience 14(6), 503–507 (1984)
24. Griffith, A.: GCC: the complete reference. McGraw-Hill/Osborne (2002)
25. Ireland, A.: On the Scalability of Proof Carrying Code for Software Certification. In: Proc. Workshop on Software Certificate Management, November 8, pp. 31–34 (2005)
26. Pfenning, F., Schürmann, C.: System Description: Twelf - A Meta-Logical Framework for Deductive Systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999)
27. Appel, A.W.: Foundational proof-carrying code. In: 16th Annual IEEE Symposium on Logic in Computer Science (LICS 2001), pp. 247–258 (2001)
28. Necula, G.C., Lee, P.: The design and implementation of a certifying compiler. SIGPLAN Not. 33(5), 333–344 (1998)
29. Mobius, Public, Deliverable D4. 1: Scenarios for Proof-Carrying Code, FP6-015905, Information Society Technologies (2006)
30. Narizzano, M., Pulina, L., Tacchella, A.: Report of the third QBF solvers evaluation, Journal of Satisfiability. Boolean Modeling and Computation 2, 145–164 (2006)
31. Barthe, G., Crégut, P., Grégoire, B., Jensen, T., Pichardie, D.: The MOBIUS Proof Carrying Code Infrastructure. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 1–24. Springer, Heidelberg (2008)
32. Chlipala, A.J.: Implementing Certified Programming Language Tools in Dependent Type Theory. Doctoral Thesis. UMI Order Number: AAI3311660, University of California at Berkeley (2007)