Secure Computing Systems Design Through Formal Micro-Contracts

Michel A. Kinsy
Adaptive and Secure Computing Systems Laboratory
Department of Electrical and Computer Engineering
Boston University
mkinsy@bu.edu

Novak Boskov
Adaptive and Secure Computing Systems Laboratory
Department of Electrical and Computer Engineering
Boston University
boskov@bu.edu

ABSTRACT

Two enduring concepts in computer system design are abstraction levels and layered composition. The design generally takes a layered approach where each layer implements a different abstraction of the system. The layers communicate through interfaces that are designed to support functional specification of the system as a whole. Traditionally, these layers and interfaces have primarily focused on functionality and efficiency — performance, power and area. Security-related issues are often overlooked or deferred until later in the design cycle or applied as add-ons when some security features are explicitly required. The challenge with this approach is that chasing security implications of certain design decisions along the multiple layers is a complex and error-prone task. Therefore, in this work, we are introducing the notion of "security micro-contracts" or simply "micro-contracts". We propose a novel secure computer systems design approach through minimal contracts — micro-contracts — between adjacent layers. These contracts have strict structures that contain security-relevant details of each connected layer and the secure properties that have to be preserved to assure confidentiality, integrity and availability of the data of interest. Micro-contracts may be used as (i) basic formalism for proving security properties of computing systems both in the software and hardware layers and across them or (ii) run time security policy checks.

CCS CONCEPTS

• **Security and privacy → Formal security models**; **Hardware-based security protocols**.

KEYWORDS

security, secure system design, micro-contracts, formal methods, layered composition, abstract level.

ACM Reference Format:
Michel A. Kinsy and Novak Boskov. 2019. Secure Computing Systems Design Through Formal Micro-Contracts. In *Great Lakes Symposium on VLSI 2019 (GLSVLSI '19), May 9–11, 2019, Tysons Corner, VA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3299874.3319447

GLSVLSI '19, May 9–11, 2019, Tysons Corner, VA, USA

ACM ISBN 978-1-4503-6252-8/19/05.
https://doi.org/10.1145/3299874.3319447

1 INTRODUCTION

The partitioning of computing systems into layers originates from the need for abstraction. Abstraction-based design enables system designers to cope with the ever-increasing complexity of the system. One of the benefits that abstraction brings into the design fold is the principle of *separation of concerns*. This principle gives us the mechanisms to achieve modularity. We usually specify module internals separately from the interfaces between modules through which the communication flows in order to implement desired functionality of the whole system. Interfaces tend to be minimal and clear in order to allow for different internal implementations of adjacent layers. However, the abstraction has also failed to establish strong security guarantees across these levels [5].

In the process of defining the interfaces, some details of the design that are considered non-essential to the functionality of the system are abstracted out, and are not necessarily preserved across the interfaces. This abstraction technique has traditionally worked. In fact, it is adopted across the full computing system stack, from instruction set architectures (ISAs) and memory subsystems to high level application programming interfaces (APIs). However, this abstraction process sometimes removes or misses important contextual or security-related information. Essentially, the process may create semantic gaps or widen them.

Through the analysis of known security vulnerabilities, one can distinguish three major classes. In this work, we use these classes to illustrate the space of possible vulnerabilities and to introduce the notion of "micro-contracts".

One class of security vulnerabilities is caused by *human error*. We usually refer to this class of vulnerabilities as security bugs. Security bugs happen when a programmer or designer of the system makes design mistakes mostly due to the weight of inherent system's complexity. In this case, security threats normally can be precluded by applying existing well-known mitigations, e.g., best software/hardware development practices. This class of vulnerabilities happens in a variety of real-world cases such as wrongly ordered pointer dereferencing causing privilege escalation in Linux kernel [12] and poor application of the *separation of concerns* principle in OpenSSH library that allows for connecting unauthorized clients to the server [11]. This ubiquitous class of security vulnerabilities encompasses a large portion of the vulnerabilities in publicly available databases such as Common Vulnerabilities and Exposures (CVE) and National Vulnerability Database (NVD). Provided that vulnerabilities are correctly documented and timely published, needed solutions are usually developed in a tolerable time frame. These solutions may require diverse set of actions including

design changes but normally do not rely on massive abstraction modifications.

The second vulnerability class stems from *misuse of the existing abstractions*. For example, a programmer may unintentionally establish its security guards on the basis of optimization-unstable code in C/C++ [15]. This may happen when a security-related logic is on a control flow path removed through an optimization. For instance, the correctness of a compiler optimization is judged in accordance with functionality and performance, not necessarily its security. In other words, when a piece of code is pruned by the compiler, program's functionality is guaranteed to remain the same while security properties as intended by the programmer may be violated. Consequently, a program that expresses some security guard in its source code may not contain the same guard in its binary. Similar security vulnerabilities may also happen with other programming languages and runtime environments with more complex and dynamic behaviors like Java [8].

Both vulnerability classes — i.e., caused by human errors and by abstraction misuse — may be partially or completely mitigated by applying static analysis techniques. The human error vulnerability class is straightforward and its resolution has been the object of many research efforts. The case of abstraction misuse is more subtle to analyze. However, there are known common abstraction misuse patterns that can be described and guarded against within the abstraction itself. Thus, some vulnerabilities within this class can be partially or completely mitigated. For example, according to the C programming language standard, there are 77 examples of undefined behavior in the core language description [4]. In general, if there is a known list of abstraction misuses, one can erect guards as part of the abstraction construction to prevent abuses, and consequently, security holes. For instance, at the programming language level, through its compiler, one can detect "negative" semantic constructs and add rewrite rules to cover cases of undefined behavior [3]. One simple approach is for the compiler to halt the compilation process when it detects an undefined behavior and alert the programmer. Using the compiler's error message, the programmer can rewrite their program to retain desired security-related properties.

The third class is the security vulnerabilities *inherent to the abstraction* itself. These vulnerabilities are generally hard or even impossible to solve without significant redesigning of abstraction. Even if the designer uses the abstraction in a correct and consistent manner, security threats are not neutralized. One such a case is speculative execution side-channel based attacks [7]. Speculative execution timing side-channel based attacks may be considered an inherent security vulnerability tied to the abstraction view that the programmer and the compiler have of the processor. Specifically, this view is an isolated, sequential execution of the program statements/instructions as opposed to a pipelined, speculative, multi-threaded, multi-program execution environment.

The CPU micro-architecture implements the abstraction given through the ISA on one end, while for example compilers implement the same abstraction on the software side. By the virtue of abstraction both layers have freedom to implement their internals independently as long as the abstraction itself remains intact. For example, CPUs minimize time per cycle and cycles per instruction abstracting out micro-architectural features such as speculative execution and caches. These features are usually not exposed through the ISA as the interface corresponding to the abstraction. That is, an adjacent layer that uses ISA (e.g., a compiler) does not distinguish loads and stores that cause a cache hit from those which cause a cache miss. The properties of instructions other than their functionality are normally absent from the abstraction. One of such absent properties that an attacker may use to obtain sensitive information is timing. An attacker may measure the time difference of cache hits and cache misses to infer the data. Since timing of the instructions is not described in the ISA and since an attacker can use this information to affect the confidentiality of the system, we consider this class of vulnerabilities as inherent to the abstraction. However, the three classes are not mutually exclusive and a single vulnerability may belong to more than one class.

2 RELATED WORK

Various approaches have been used in the study of security property preservation targeting the different layers of computing abstractions. Some approaches focus on guarding the compilation process from attacks in low-level co-linked components [13]. Other researchers have tackled security implications of undefined behavior in programming languages by also taking into account the internal machine states to bridge the gap between security and the correctness. For example, D'Silva et al. [2] study security effects by extending source machine, a formalism used in studying compiler correctness. Some other approaches [3, 15] consider all the undefinedness in C as potential security holes and give rewrite rules to catch all the standard-prescribed undefined behaviors or use other techniques to discover it.

Multi-layer approaches are particularly interesting from the perspective of this work. In some approaches, the authors chose to implement a subset of C [1] on top of capability hardware [16] and investigate the preservation of security properties through memory object model [10]. With other techniques the focus was more on enforcing end-to-end security policies starting from high-level languages as described by Sabelfeld and Myers [14]. Mcilroy et. al. [9] recently took on the problem of understanding microarchitectural side channels and the formal analysis of their security implications.

3 GENERAL FRAMEWORK FOR MICRO-CONTRACTS

Micro-contracts are designed as a unified framework to tackle all three classes of vulnerabilities in a formal and security-centric manner. The solutions for known vulnerabilities that we are aware of normally mitigate each vulnerability using a relatively unique method. However, they do not describe security implications of the mitigations themselves. The solutions are normally demonstrated on minimal examples contained in vulnerability databases. Their implementation in complex systems usually requires numerous other changes whose security implications are studied separately from the vulnerability being mitigated.

For example, compilers rely on ISA of the underlying processor. They usually perform various optimizations without taking into account implementation details of instructions nor the high level intent of the programmer. One such optimization is *strength*

reduction which can modify two previously equally time intensive branches to two branches that have observable difference in execution time. In this case, the difference is caused by compiler's decision to replace expressions with static values that do not consume any execution time or to change time complexity of the calculation. Further, an attacker may time branches and infer security sensitive information [2]. Even though some of the security implications of this kind may be tackled by disabling certain optimizations in the compiler, it may also cause an unacceptable runtime overhead. Having these constraints in mind, the programmer is forced to make an ad-hoc performance-security trade-off. The programmer can apply the separation of concern principle and separate critical code portions from the rest of the program. When the separation into critical functions (or even files) is done, the programmer can use specific compiler options to suspend particular optimizations on the critical code portions. In practice, this process usually requires a programmer that understands not only the motivation for the particular compiler optimization but also its internal implementation in the specific compiler. Furthermore, the programmer has to understand details of the compiler's infrastructure to properly apply optimization restrictions.

The approaches similar to the previous one may be useful as ad-hoc solutions. However, the approaches may also be considered as intermingling multiple abstractions through bypassing the high level interface of the programming language imposed by the standard and implemented by the compiler. The programmer is expected to explicitly use compiler's internals in order to achieve the desired program behavior. This poses a violation of computing system modularity and thus impedes reasoning about the complex systems' behavior. Consequentially, tracing security implications of certain design decisions becomes harder.

We propose micro-contracts, a strictly defined, minimal and modular framework for ensuring the security of layered computing systems. Micro-contracts are security-centric in the sense that they consider security of the whole system as the strict boundaries imposed to the functionality. This framework is also data-centric in the sense that it focuses on expressing desired security-related properties of the data instead of describing the internals of security preserving functions of the system's parts.

3.1 Design

Micro-contracts are designed as a framework for security analysis and policy enforcement in complex computing systems divided into layers. Policies are formed on the basis of the concrete attacks being analyzed. A general overview of the micro-contracts framework is given in Figure 1.

A micro-contract represents a security-related agreement between two adjacent layers in the computing stack. The two adjacent layers are represented as CSL_i and CSL_{i+1} and may denote e.g., high level programming language and its compiler, respectively. CSL_i describes higher level intents of the programmer while the compiler produces the binary image that is executed. In general, *CSLs* are given through their abstractions such as a programming language standard or an ISA of a CPU.

Security policies from layer CSL_i are translated in CSL_{i+1} by some function $f : SP_k^{CSL_i} \rightarrow SP_k^{CSL_{i+1}}$. As we explain in §4, some

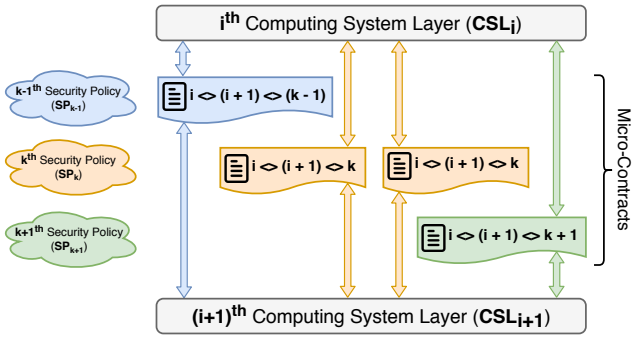

Figure 1: Overview of the micro-contracts framework.

properties of SP_k may be lost after the application of function f while some other properties may be added. To preserve security properties originally expressed in SP_k while it gets passed down the layers, one has to characterize the translation (f). A micro-contract is the tool that helps us to describe the translation with regards to the security property that we want to enforce. Additionally, it offers the mechanism for the analysis of the security implications of the design decisions made on different layers of the computing system. A security policy may be enforced by many micro-contracts while a single micro-contract should be related to only one security policy. Relations between security policies should be studied through the composition of micro-contracts. Normally, we have multiple micro-contracts between two layers. Micro-contracts must conform to the strict structure given in Figure 2. This structure is logical rather than syntactical and may be implemented in different syntaxes as well as in the form of ontologies. For the sake of simplicity, in this work we give the examples using a language similar to YAML.

```
information ::= "information: " (info-descriptor
                                 | "Any") ;
info-descriptor ::= {ID primary-concept} ;
primary-concept ::= "primary concept:"
                    ("confidentiality"
                      | "integrity"
                      | "availability") ;
involved-layers ::= "involved layers:" {layer} ;
layer ::= "compiler"
          | "programming language"
          | "microarchitecture" ;
layer-concepts ::= "layer concepts:" {layer-concept} ;
layer-concept ::= ID
                  "belongs to:" layer
                  "defined as:" definition
                  "criteria:" criteria ;
definition ::= (* available definitions *) ;
criteria ::= (* criteria expressions *) ;
```

Figure 2: Partial general micro-contract grammar in extended Backus-Naur form.

The specification of micro-contracts has three major parts: *information*, *involved layers* and *layer concepts*. The information part is aimed to express the data that is compromised in the attack for which the mitigation is enforced within the security policy. Each part of the information can be described separately. The primary

security feature that may be violated must be listed for each part. Available primary information security features are confidentiality, integrity and availability. If for any reason information part is not applicable for the micro-contract, "Any" may be used. In §4.2 we give an example of such a micro-contract.

Involved layers are possible concretizations of *CSLs*. For the sake of grammar brevity we do not list all the possible layers in the computing stack. There is also an additional rule imposed on this micro-contract part: *two layers listed in a micro contract have to be adjacent in the computing stack.* Only two layers are allowed, when there is the need for connecting more then two layers a composition should be used.

Layer concepts are the central part of a micro-contract. Each layer concept represents a layer abstraction's part crucial for the enforcement of the security policy. Layer concepts are internally distinguished using identifiers. Each layer explicitly lists the abstraction that it belongs to. It also declares the way it is defined in the native layer. Some important kinds of definitions are: property (specific ability of a layer), internal (internal layer concept), simple data (numbers, strings etc.) and auxiliary definitions. The last kind represents the predicates that have to be defined by the author of the micro contract for the involved layer. In the examples in §4 we use only one auxiliary definition — `grammar`. In one case it represents the grammatical rule in programming language C while in other it represents grammatical rule in RISC-V ISA.

The criteria part of layer concepts is given in the form of condition that is used to decide whether particular layer concept must be enforced. It may contain auxiliary predicates that have to be implementable in the native concepts of the corresponding layer. For example, in §4.2 we use `contains(expression, variable)` auxiliary predicate from the compiler layer. Since all compilers have the notion of expressions and variables, the auxiliary predicate is easily implemented.

3.2 Composition of Micro-Contracts

Often, it is not enough to track the security implications only to the adjacent layer but also deeper along the computing stack. In this case, micro-contracts provide the mechanism of multi-layer composition. This kind of composition is aimed at tracking the enforcement of security policies along the computing system stack.

As shown in Figure 3, we connect two or more layers by inserting at least one micro-contract between each two layers. The mapping represented by the arrows in Figure 3 is now:

$$f : SP_k^{CSL_{i-1}} \rightarrow SP_k^{CSL_i} \rightarrow SP_k^{CSL_{i+1}}$$

However, not all the micro-contracts can compose so as to preserve desired security property. Thus, we define the following rule:

> *Two micro-contracts can compose iff the conjunction of their criteria is satisfiable.*

Criteria of a micro-contract, denoted as MC_j^{crit}, is given as a logical expression $MC_j^{crit} \equiv \alpha \rightarrow (\beta \wedge \gamma)$ where α, β and γ represent constituents of the criteria expression in MC_j^{crit}.

Assume that we compose MC_j with MC_{j+1} and that:

$$MC_{j+1}^{crit} \equiv (\alpha \rightarrow \delta) \wedge (\delta \rightarrow (\neg\gamma \wedge \beta))$$

Then, the criteria of composition becomes:

$$MC_j^{crit} \wedge MC_{j+1}^{crit} \equiv \alpha \rightarrow (\beta \wedge \gamma) \wedge (\alpha \rightarrow \delta) \wedge (\delta \rightarrow (\neg\gamma \wedge \beta))$$

This composition is satisfiable for $\langle \alpha, \beta, \gamma, \delta \rangle := \langle \bot, \top, \bot, \top \rangle$ meaning that MC_j can compose with MC_{j+1} so as to preserve security policies expressed by the micro-contracts. If the composition criteria were unsatisfiable that would indicate that two micro-contracts cannot compose in the security-preserving manner.

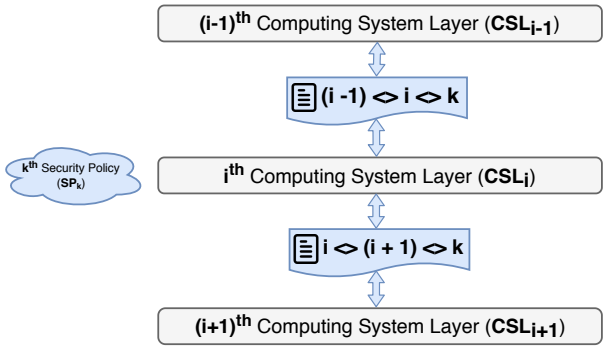

Figure 3: Multi-layer composition of micro-contracts.

4 ILLUSTRATIVE EXAMPLES

To illustrate the spirit of micro-contracts we give two examples of their usage on two distinct layer joints. The first one is a timing side channel attack opened by the strength reduction compiler optimization [2, III C]. This example describes a micro-contract that lies between high level code in programming language C and its compiler. The second example is Spectre1.1 speculative buffer overflow attack [6]. For the sake of illustration we present a hypothetical micro-contract that tackles Spectre1.1 on the joint between ISA and microarchitecture.

4.1 A Strength Reduction Micro-Contract

A simple example of strength reduction is implementation of multiplication using bit shift operations. For example, an expression that contains multiplication by 2 can be reduced to a functionally equivalent yet less processor intensive expression that uses left shift by one. Thus, expressions in the form `expr*63+23` can be rewritten as `(expr<<6)-expr+23` for any `expr` representing an integer.

Even though strength reduction can substantially improve the performance of various arithmetic expressions especially in loops, it may have hazardous security implications. Sometimes, programmers intentionally try to make branches equally computationally intensive in order to hide the information about what branch has been taken. In this case, strength reduction may open a timing side channel through which sensitive information can be leaked [2].

In the Figure 5 we give an example of a micro-contract to tackle the vulnerability incurred by strength reduction optimization in the compiler. This micro-contract pertains the information contained in the critical variable. The primary security feature that is compromised in this kind of attack is the confidentiality of the cryptographic key stored in the critical variable. In practice, we want to maximally hide the key from the attacker. Since attacker

may utilize the time difference between branches, it may be better to leave the branches that handle the key in the original form expressed by the programmer. We assume that the programmer is aware of the timing side channels and is able to apply some of the known measures to make branches as closely computationally intensive as possible. Having said that, the security analyst may decide to use the following policy:

> *The programmer should declare keys using the* KEY_DECL *pragma. Compiler is not allowed to apply strength reduction on branches that contain computation that involves keys.*

In Figure 4 we show C code snippet that uses the KEY_DECL pragma on declaration of variable that holds the key at line 2. The programmer's idea is to avoid the timing difference between branches in line 5 and 7 by using one multiplication and one subtraction in both cases. Such written code should not exhibit any timing differences between the two branches.

```
1  int crypt(int x) {
2      #pragma KEY_DECL
3      int key = 0x23;
4      if (x == 0xDEAD) {
5          key = key * 7 - 5;
6      } else {
7          key = 2 * 17 - 3
8      }
9      return key;
10 }
```

Figure 4: An example of a micro-contract-abiding C implementation of a toy cryptographic function.

```
information:
    - critical_variable
        primary concept: confidentiality
involved layers:
    - programming language
    - compiler
layer concepts:
    - critical_variable:
        belongs to: programming language
        defined as: grammar("#pragma KEY_DECL")
        criteria: Always
    - expression:
        belongs to: compiler
        defined as: internal
        criteria: Always
    - strength_reduction:
        belongs to: compiler
        defined as: property
        criteria:
            if contains(expression, critical_variable)
               then absent
               else present
```

Figure 5: An example of a micro-contract for mitigating timing side channel attacks allowed by the strength reduction compiler optimization.

This is an example of the micro-contract between the programming language and the compiler. Such a contract may be implemented between e.g. C11 programming language and the LLVM compiler.

The layer concepts involved in the micro-contract are the critical variable, condition and strength reduction. The critical variable belongs to the programming language. This practically means that the designers of the micro-contract decided to extend C programming language with a technique for marking critical variables. The idiomatic way of doing so in C is by defining a pragma which will then have a generic pragma syntax `#pragma KEY_DECL`. `expression` is the primitive concept that belongs to compiler and is defined as its internal. This means that a compiler internally contains the notion of expression that can be given to disposal of micro-contracts. `Strength_reduction` is the concept that belongs to compiler and is defined as property. This means that the micro-contract uses it as the capability of the compiler that can be instrumented. The way a micro-contract instruments the property is given by the criteria. In this case, the micro-contract turns off strength reduction by declaring it absent in the cases when the expression contains the critical variable.

The mechanism for checking whether some variable is contained in the expression is normally present in the compiler's framework because it is needed even in the fundamental scope resolution.

It is important to mention that the vulnerabilities that are incurred by compiler optimizations can involve other optimizations such as common subexpression elimination or peephole optimizations. To tackle these vulnerabilities, the security analysts may decide to extend their policy and design one contract that describes the extended policy. However, the micro-contracts framework is designed to embrace minimal contracts and advocate the approaches with their composition.

4.2 A Spectre1.1 Micro-Contract

Spectre1.1, as introduced by Kiriansky and Waldspurger, is a Spectre [7] variant that uses speculative store instruction execution in modern superscalar processors to induce buffer overflows. Speculative store may be utilized to cause an out-of-bounds write that overwrites the return address of the function in which a vulnerable code portion resides. From that moment on, the attack resembles return-oriented-programming and can even invoke the critical part of Spectre1.0 gadget if present to leak sensitive information.

However, an important prerequisite of Spectre1.1 is pointed out by its authors. The speculative window must be wide enough to admit all the payload gadgets plus the instructions that lie between the vulnerable conditional branch and the indirect return instruction that is being attacked. If the wrong prediction is discovered before the return, the corrupt data needed for the attack will be invalidated and the attack will fail. For our purposes, we will assume that the security analyst that works on mitigating the critical vulnerability decided to approach the problem using the following policy:

> *Branches that lead to basic blocks with small number of instructions are more vulnerable to Spectre1.1. Instruction set defines* sb* *(short branch) instructions for all the branching conditions.* sb* *branches are branches that do not allow speculation window wider than* IC_{short} *instructions.*

Since the policy is given in the terms of ISA and does not refer to any higher layers (e.g. compiler), the micro-contract corresponding to the policy will lie between ISA and the microarchitecture. There is a multitude of such micro-contracts that may be constructed. We are giving one in Figure 6 as an illustrative example.

Layer concepts involved in the micro-contract from Figure 6 are maximal number of instructions in short branches (`IC_short`), short branch itself and speculation. `IC_short` belongs to ISA. This practically means that ISA is being extended to include this information. It is defined as a number. Since this is primitive concept needed for more complex ones, it should always be enforced by the micro-contract. `Short_branch_eq` is a more complex concept present on the level of ISA. It is defined grammatically as a regular instruction similar to `BEQ` (branch if equal). For the purpose of this example, we use RISC-V-like instruction syntax. However, the concept is general and supports any concrete ISA for which the author can provide corresponding grammar definition. Finally, this concept has to be enforced by the micro-contract when instruction count of the branch is smaller than the maximal number of instructions in short branches.

The `speculation` part represents the processor's ability to speculatively execute instructions. It is clearly a property of the microarchitecture itself. The criteria under which the micro-contract enforces speculation to be turned off is `short_branch_eq`. When `short_branch_eq` is enforced on the ISA level then speculation is suspended. This particular part of the micro-contract illustrates the logical junction between two abstractions without intermingling their internals.

It is also important to note that `sb*` is not completely covered by the micro-contract in Figure 6. However, the criteria for all the branch instructions is considered to be the same. Thus, extension of this micro-contract to cover all the branches would just add simple grammatical definitions for the other five RISC-V branch counterparts (`BNE.s`, `BLT.s`, `BGE.s`, `BLTU.s` and `BGU.s`) while other parts would remain the same. For the sake of convenience, future versions of the micro-contracts framework will provide the support for concise expressions of repeating parts. One of the possible approaches is inheritance of the layer concepts. However, the ontological information contained in a more concise version of the micro-contract is exactly the same as in its extended version.

Additionally, we may need to construct a finely-grained mechanism for tackling security implications of Spectre1.1 further up the computing stack. That is, we may be interested in analyzing the security of our system through reasoning about the high level concept such as cryptographic keys given to the system as variables of a program. To do so, we need to design the chain of micro-contracts so as to cover upper layers such as e.g. assembler, linker, compiler and the programming language itself. The micro-contracts framework embraces writing multiple micro-contracts that chain to each other all the way down to the microarchitecture. Therefore, we eliminate a portion of the complexity of non-trivial computing systems that impedes our reasoning about information security.

5 CONCLUSION

In this paper we introduced the micro-contracts framework as a formal, minimal and modular methodology for enforcing security policies over multiple layers of computing systems. We discussed the building blocks of a micro-contract as well as the internal structure. Afterwards, we described multi-layer composition of multiple micro-contracts and introduced the concept of composability. Finally, we demonstrated the usability of the introduced concepts through the design of the micro-contracts among different layers.

```
information:
    - Any
involved layers:
    - ISA
    - microarchitecture
layer concepts:
    - IC_short:
        belongs to: ISA
        defined as: number
        criteria: Always
    - short_branch_eq:
        belongs to: ISA
        defined as: grammar("BEQ.s rs1, rs2, imm")
        criteria: instruction_count() > IC_short
    - speculation:
        belongs to: microarchitecture
        defined as: property
        criteria: if short_branch_eq then absent
                                      else present
```

Figure 6: An example of micro-contract for mitigating Spectre1.1 attack.

We focused on guarding against timing side channel attacks introduced by strength reduction compiler optimization and Spectre1.1. We envision micro-contracts as a sound methodology for studying cross-layer security implications of both future attacks and their mitigations.

REFERENCES

[1] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. 2015. Beyond the PDP-11. *ACM SIGARCH Computer Architecture News* 43, 1 (Mar 2015), 117–130.

[2] Vijay D'Silva, Mathias Payer, and Dawn Song. 2015. The Correctness-Security Gap in Compiler Optimization. *2015 IEEE Security and Privacy Workshops* (May 2015).

[3] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the undefinedness of C. *ACM SIGPLAN Notices* 50, 6 (Jun 2015), 336–345.

[4] ISO/IEC JTC1/SC22/WG14. 2011. *Information technology - Programming languages - C.* ISO/IEC 9899:2011. International Organization for Standardization.

[5] M. A. Kinsy, S. Khadka, M. Isakov, and A. Farrukh. 2017. Hermes: Secure heterogeneous multicore architecture design. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 14–20.

[6] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. arXiv:cs.CR/1807.03757

[7] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[8] V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14 (SSYM'05)*. USENIX Association, Berkeley, CA, USA, 18–18.

[9] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. arXiv:cs.PL/1902.05178

[10] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the depths of C: elaborating the de facto standards. *ACM SIGPLAN Notices* 51, 6 (Jun 2016), 1–15.

[11] MITRE. 2009. CVE-2009-1897.

[12] MITRE. 2018. CVE-2018-10933.

[13] Marco Patrignani and Deepak Garg. 2017. Secure Compilation and Hyperproperty Preservation. *2017 IEEE 30th Computer Security Foundations Symposium (CSF)* (Aug 2017).

[14] Andrei Sabelfeld and Andrew C Myers. 2003. Language-based information-flow security. *IEEE Journal on selected areas in communications* 21, 1 (2003), 5–19.

[15] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards optimization-safe systems. *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13* (2013).

[16] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, and et al. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. *2015 IEEE Symposium on Security and Privacy* (May 2015).