A Rest Stop on the Unending Road to Provable Security (Transcript of Discussion)

Virgil D. Gligor(✉)

Carnegie Mellon University, Pittsburgh, USA
virgil.gligor@gmail.com

The title of this paper is a spoof on Butler Lampson's assertion that there is no resting place on the road to perfection [1]. While I don't disagree with his assertion, the road to provable security is not exactly a road to perfection, so we can expect some rest stops. In this presentation I will first define what I mean by a *rest stop*, then I will argue that rest stops are hard to find, and finally will explain how to find one in one case.

Let's start with a question Fred Schneider asked in his blueprint for a science of security: "Is it ever possible to add defenses and transform one system into another, where the latter system requires weaker assumptions about the components being trusted?" [2] In essence, this question is about adding *trustworthy reductions* (i.e., defenses) to decrease the security liability of the trust assumptions required by a security property, or more. Let's suppose for the moment that the answer to this question is "*it's always possible*." Then, in principle, we could compose reductions repeatedly until we remove *all* trust assumptions. Thus, we could create trust *unconditionally*, or more accurately trustworthiness, for whatever security property we obtain at the end of the composition. (This exercise is hypothetical: Schneider's paper does not imply that trust creation is practical.)

A Hypothetical Example. Suppose that we define program partitions in an untrusted system. Then we remove the assumption that information does not flow between different partitions by adding a trustworthy *separation kernel*, which reduces a simple information-flow isolation model to a system with *isolated* partitions; see John Rushby's 1981 proposal [3]. Now if we connect this system to a network, we still have the remaining assumption that the network maintains flow isolation between network partitions. However, if the answer to the question above is "*it's always possible*," then there *must be* a way to add another trustworthy reduction (or more) to this system and remove this remaining assumption. Suppose that we add another trustworthy reduction proposed by Rushby, namely a *red/black separation* [4]. For example, messages that originate from a red (classified) partition are split into two parts: the message data, which are encrypted in a separate partition, and the message header, which is sent to another separate partition to reduce the amount of information that can be leaked by header modulation (e.g., destination addresses, packet length, time between transmissions) to some acceptable level, possibly zero. Then a

J. Anderson et al. (Eds.): Security Protocols 2019, LNCS 12287, pp. 233–241, 2020.
https://doi.org/10.1007/978-3-030-57043-9_22

fourth partition combines the controlled header content with the ciphertext of the message data, and releases the resulting black (unclassified) message to the network. In effect, the four partitions and the unidirectional flows between them implement a red/black reduction which separates network flows between isolated partitions and removes the remaining trust assumption.

Conditional Reductions. Now recall Lampson's metaphor that "security is fractal: each part of it is as complex as the whole;" see the above-cited paper. Clearly, if this is true, neither of the previous two examples of reductions can be unconditional, as their trustworthiness *must be* conditioned on some other complex assumption. For example, to be trustworthy, the separation kernel must be formally verifiable, which implies that it must always be small and simple. However, this condition does not always hold in practice: in 2010, the Information Assurance Directorate of the NSA argued that high assurance for (i.e. formal verification of) separation kernels is inappropriate for *commodity* workstations due to their complexity [5]. Furthermore, the cryptographic library of the encryption partition used for the red/black separation reduction must also be formally verifiable. However, this is conditioned by hardness assumptions, specific bounds on the adversary computation power, and often hardware security modules. Since the trustworthiness of these reductions is conditional, it is impractical to create trust in network isolated partitions by composing them as in our hypothetical example.

If security is indeed fractal, the answer to Schneider's question above is always negative. His conjecture that "trust cannot be created, it can only be relocated," [6] supports this, and the composition example above illustrates the conjecture. First, we relocated trust from our system with isolated partitions to the first reduction, namely to the separation kernel, whose trustworthiness was assumed, since it is not always formally verifiable. Second, we relocated trust from the red/black separation network to the hardness assumptions made for the formal verification of the cryptographic libraries. The impracticality of creating trust unconditionally was also illustrated by an earlier paper where Ivan Damgård pointed out that, in cryptography, reduction proofs only relate security properties to unproven conjectures [7]. Finally, in his commentary to my talk at SPW-22, Dieter Gollmann came up with what I call the *Gollmann complaint* [8]: "reductions often assume that something unknown cannot happen" – a very succinct statement of the general problem.

Simon Foley: I guess the question I have is, what do you mean by reduction, because we're talking about information flow for security properties. If what we mean by reduction is, say, trace semantics of the system, then the reduction is the removal of certain traces. In that case, the composition is not going to preserve security properties because sometimes to achieve a security property we want to inject additional traces, or additional behaviours.

Reply: Yes, one can view my reduction examples as removals of certain traces. Note that I was careful to pick reductions that do compose [9]. That is, if one

has isolated flow partitions, one can construct the red/black separation out of four partitions and unidirectional channels.

However, you're absolutely right that, in general, not all reductions can be composed to achieve a desired security property the way I did here.

Simon Foley: So a reduction is, in simple terms, removal of traces, and what you're producing in Fred Schneider's terms, is enforceable mechanisms or approximations of information-flow properties.

Reply: Certainly, one can view my example reductions this way.

Unconditional Reductions. The possibility of creating trust in a security property implies the existence of *unconditional* reductions. A reduction is unconditional if its efficient, trustworthy execution uses *no* secrets, *no* trusted hardware modules, and *no* bounds on the adversary power. For example, the trustworthy verifier implementing such a reduction would only need truly random bits and the specifications of a device's hardware. Finding truly random bits is not a major hurdle in practice. If the bandwidth of the random bit generator need not be very high (e.g., a bit per second), one can even build his/her own true random number generator by capturing simple quantum effects. Otherwise, one can purchase a commercially available quantum random number generator; e.g., from ID Quantique. The harder part is finding correct and complete device specifications. Note that this is *always* required: without such specifications, one cannot expect solutions to either security or cryptography problems. Instead, one can only expect surprises. Finally, the trustworthiness of the small and simple verifier must be proven and the proofs must not depend on any unverified computation.

What Is a Rest Stop? The existence of unconditional reductions is significant in security. Unconditional reductions imply that the security properties obtained have no dependency on conjectures whose veracity is unknown or on unknowable behaviours of individuals who install secrets in one's system. They offer provable advantage to a defender over any adversary, and outlive technology advances; e.g., they are useful post quantum computing. In effect, an unconditional reduction represents a practical *rest stop* on the unending road to provable security. Once we find a *rest stop* for a security property the property is unconditionally proved, and then we can move on to prove other security properties. I must admit that I do not know of any *rest stop* in software security or cryptography other than the one I describe in the following example.

A Rest Stop Example. Let a small and simple verifier machine be connected via a local bus to an external device that may contain *persistent malware*. The verifier wants to establish that the device contains all and only the content it initialized in the device (i.e., it establishes *root of trust*), and hence the device is malware free. The verifier knows the device's hardware specifications ranging from the ISA, register set (R) primary memory (M) caches, virtual memory,

TLB, pipelining, multiprocessors, etc. Furthermore, the verifier has a source of true random numbers. Then, the verifier initializes the device to a family of concrete space-time optimal computations $C_{m,t}$, and challenges the device to execute a program C_{nonce} in memory (M, R), whose optimal size m, and time t on a *nonce* constructed from the true random numbers. If $C_{m,t}$ is second pre-image free and the correct result $C_{nonce}(M, R)$, which is unpredictable, arrives in time t, the verifier concludes that the device is malware-free *efficiently*; i.e., in minutes [10]. Note that the adversary who controls the device malware has unbounded power but is remote. Thus it can only modify a device's software and firmware but not its hardware. Furthermore, it cannot access or foresee the true random numbers used by the verifier to construct the *nonce*.

Trustworthy Verification. The verifier has three trustworthy security properties. First, it leaks neither the random numbers to the device (malware) *before* it sends the *nonce* nor the result $C_{nonce}(M, R)$ and optimal t before it receives the untrusted-device response; e.g., it obtains and verifies the correct result $C_{nonce}(M, R)$ and optimal time t *after* the device returns its response to the verifier's *nonce*. These two *no-leakage* properties can be obtained by standard information flow tracing of verifier code; e.g., by taint analyses of binary code. Second, the verifier knows the device specifications and can either simulate this computation on a malware-free simulator or obtain the result and its timing from another similar device that is known to be malware-free. Third, the verifier can measure the device computation time in a provably correct manner; e.g., it can *verifiably* disable caches/TLBs, virtual memory, and *verifiably* set clock frequency.

In short, the verifier can be trusted to perform an unconditional reduction from the concrete space-time optimal computation to the device's root of trust property.

Jonathan Anderson: So the protocol is using the *overt* channels that the device is presenting to the verifier, but that doesn't necessarily prove that there is an absence of *covert* channels, right?

Reply: Yes. It must be proved that the verifier does not leak other information than the *nonce* to the device. Recall that to have a covert channel, one has to have Trojan horse, or a spy, code on the verifier that colludes with the receiver-device malware; i.e., it encodes information and sends it to the colluding device malware. There is no such thing on a *trusted* verifier.

Jonathan Anderson: But is that just because the random bits are never shared with the device?

Reply: Yes, but also because the verifier is trustworthy; e.g., one can prove that the random bits enter the local verifier, but not the device, and exit as the *nonce*, by information-flow tracing.

Jonathan Anderson: Right. So I guess one can prove that the device faithfully executes a function, but this doesn't necessarily prove what else the device is doing. There could be other behaviours.

Reply: The device cannot do anything else, since the computation is proven to be concretely space-time optimal on that device. And the device's hardware specifications are assumed to be correct.

Jonathan Anderson: Right. So, I guess this seems to put a lot of importance on having correct device specifications, but isn't the problem of verifying hardware devices even more challenging than the problem of verifying the correct result and its timing?

Reply: No, because the correct hardware specifications come from the manufacturer. And if they are incorrect, one cannot talk about either software security or cryptography solutions. One can only expect surprises.

Jonathan Anderson: That's true, but supply-chain attacks are possible, right?

Reply: Yes, but the protocol takes care of those, as I will show shortly.

Fabio Massacci: I have a question about $C_{m,t}$. This computation must use the full extent of the device memory. So if one uses only a subset of the memory, the computation is no longer going to be space-time optimal. Thus every time one executes $C_{m,t}$, one uses all available device memory. Right?

Reply: Exactly.

Martin Kleppmann: If you have the local verifier that's already sitting on the bus, why can't the verifier just read all of the memory of the machine and check that it has the expected state? Why do you need all of the complexity of these optimal $C_{m,t}$ algorithms?

Reply: One needs these computations because otherwise device malware can simulate the contents of the device memory and send them to the local verifier. Then the verifier will never know that it read the output of the malware simulator.

Martin Kleppmann: But you are, at the same time, assuming that your device specification accurately represents the speed at which the CPU can execute the algorithm, for example.

Reply: Correct. Accurate device specifications are always required.

The verifier also needs to use a space-time optimal computation because otherwise malware could hide in persistent memory and survive power cycles. Imagine that the device happens to be a controller (i.e., a disc controller, or a NIC controller, or a DMA controller, or a manageability engine controller), which has flash memory. That memory could contain malware that is inserted there by the *supply chain* [11,12]. For example, an on-line supplier-generated patch can be surreptitiously interdicted by an adversary who inserts malware into the controller's persistent (e.g., flash) memory. Running the space-time optimal computation either detects the presence of malware and other unaccounted content or returns a correct and timely result. In contrast, if the malware is in non-persistent (i.e., volatile) memory, all the verifier has to do is to turn off the device and the malware goes away.

Why is it hard to find such a rest stop? There are two reasons for this. First, complexity theory lacks examples of computations that are non-asymptotically space-time optimal in adversary execution on the device's instruction set architecture (ISA) rather than on abstract machines; e.g., on Turing Machines or algebraic computation models [13]. Second, even if such a computation is found, adversary malware can be powerful enough to violate any control flow integrity of the verifier-device protocol and exploit availability of multiple device controllers; e.g., space-time optimality in sequential execution does not necessarily hold when multiple concurrent device controllers execute different optimal computations.

For example, Horner's rule for a degree-d polynomial evaluation is known to be *uniquely optimal* in infinite fields using $2d$ operations: d multiplications and d additions. However, is is *not* optimal in finite fields or on any realistic ISA or when the device is controlled by an adversary. Furthermore, the verifier can send only a single input packet of $k << d$ words to represent a polynomial to the device regardless of how large d is. Otherwise, malware could pre-process the evaluation while additional inputs arrive and circumvent the evaluation bounds.

Even if the adversary cannot circumvent space-time optimality of polynomial evaluation on a specific device ISA and all attempts to use a powerful remote proxy (e.g., quantum) computer are detected, the adversary can post a future interrupt that violates the control flow integrity of the verifier-device protocol; i.e., it triggers after the optimal evaluation result is sent to the verifier and reboots a malware controlled OS image unbeknown to the verifier.

Furthermore, if the verifier is connected to multiple devices (e.g., peripheral controllers that have flash memory), each device must execute a different space-time optimal evaluation whose t scales independent of m. Otherwise, device evaluations cannot start and finish at roughly same times.

A Solution. A solution that satisfies all necessary requirements can use k-independent (almost) universal hash functions that are implemented by degree-d *randomized polynomials* using input v of $d + 1$ *log* p-bit words in length, independent of k. These polynomials, which we denote by $H(v)$, have degree d and k-independent, uniformly distributed coefficients in $\mathbf{Z_p}$. Each coefficient s_i is itself computed as a polynomial with k random coefficients and input variable $i + 1$. They are second pre-image free, so no adversary can find memory or

register words whose contents differ from the verifier's choice and pass its check, except with probability close to a random guess. To input a $H(v)$ into a device, the verifier needs to send only $k + 1$ random numbers to the device.

Randomized polynomials have non-asymptotic optimal space-time bounds; in one-time evaluation on a concrete Word Random Access Machine (cWRAM); i.e., $m = k + 22$ and $t = (6k - 4)6d$, where t scales with d independent of m. These polynomials offer stronger collision freedom properties than ordinary universal hash functions. That is, no function and input exist in $\mathbf{Z_p}$ such that the function evaluation on the input matches a given randomized-polynomial evaluation result with more than a very small probability.

Randomized polynomials enable a verifier to check the *integrity of control flow* in the code it initializes on a cWRAM device. In turn, this helps implement *verifiable* time measurements by provably disabling asynchronous events, caches/TLBs, virtual memory and stateless peripheral controllers, and by setting clock frequency and the content of the special CPU state. They also assure *bounds scalability*, which enables the verifier to establish the existence of its chosen content on the initialized device. Finally, one can show that the space-time optimality of the randomized polynomials in cWRAM can be retained on a real device ISA, such as that of a x86 processor.

Fabio Massacci: I have a question about the instruction in memory *other* than those for calculating the randomized polynomial. How and when do you execute those? Of course, you need to have other instructions in memory; otherwise, you wouldn't have malware in the first place. Right?

Reply: Yes. Please notice what is in memory at this point: some initialization and input/output code, which doesn't have to be space-time optimal, the polynomial evaluation code, and lots of constants that fill the rest of the memory. However, I do *not* have any (e.g., OS, application) code I am really interested in executing now, but what I have helps me establish the content of the system state, with high probability. After this, I can load the instructions I really want to execute in my malware-free state; e.g., I can load the programme that establishes a secure initial state. In other words, it establishes the invariant that I want to set, such as the device is disconnected from the internet.

Fabio Massacci: So this means that first you run this code to make sure that the system state is clean.

Reply: Yes. After that, I can load any other instructions I want, because the system is in a malware-free initial state that is secure. This is when I start the system; e.g., I can perform a real boot now.

Jonathan Anderson: So, this is about proving that there's no malware in firmware on devices on your computer. What about things like microcode inside of CPUs?

Reply: If it's writable, it's already included in here; if it's not, it's hardware.

Jonathan Anderson: It's sort of writeable. One can't write arbitrary things to it; one can only write signed updates to it, for example.

Reply: That would be fine. These writable registers are part of the CPU state that is captured in the input v of the randomized polynomial, $H(v)$. This part of the CPU state can be pretty large [14] as it includes registers that indicate the disabling of asynchronous events, caches/TLBs, stateless devices, and setting clock frequencies. The initialization code sets all CPUs' states. The device may also include stateless components; e.g., the GPU is code-stateless, as it does not contain persistent *code*. So if one disables it, it can no longer contain any persistent malware.

Note that the input v includes all the CPU state, which can be pretty large, as noted. If any part of it changes, as for example when malware skips the disable or set register instructions, the correct input v is modified, so the end result will be different from the one expected by the verifier.

Jonathan Anderson: My concern is that, although we implicitly trust hardware currently, we're also all aware that it could be compromised. But in this circumstance, the thing I slightly worry about is that now you're going to make stronger assumptions about the trustworthiness of the device that you're running on, because you proved something based on an assumption that you have completely accurate device specifications. And that just makes me a little bit uncomfortable. Getting device specifications, like "tell me the proprietary details of your pipeline, please," seems impractical.

Reply: I'm not worried much about the hardware security itself. Why? If somebody finds security bugs (e.g., hardware Trojans) in an unscrupulous manufacturer's hardware, the manufacturer cannot patch/remove them online, unlike software/firmware. Eventually that manufacturer will go out of business. This suggests that the hardware security problem may not be as big as one thinks, because of business deterrence. However, I am concerned about the correctness of the (security-bug free) device specifications. This can be a major problem, as you noticed.

Who may be interested in providing correct device specifications today? Some device-controller, micro-controller, medical-device manufacturers, who have to know the specifications of their devices or else they won't sell them, for legal liability reasons. However, this is not necessarily true for laptop manufacturers, for example, where one can't even get to the bus for the purpose of attaching a verifier. Eventually one will get accurate device specifications and local bus connectivity for verification/testing purposes. A place to start is automotive engineering, where one can plug verification devices into the CAN bus and check

almost everything connected to that bus. That's the kind of thing that we need to do, and for this to work we must have interfaces to plug in verifiers. We are not there yet, but we will be.

Alastair Beresford: All right, let's thank Virgil for his talk.

References

1. Lampson, B.W.: Usable security: how to get it. Commun. ACM **52**, 25–27 (2009)
2. Schneider, F.B.: Blueprint for a science of cybersecurity. Next Wave **19**(2), 47–57 (2012)
3. Rushby, J.: The design and verification of secure systems. In: 8th ACM Symposium on Operating System Principles, Asilomar, CA, pp. 12–21, December 1981. (ACM Operating Systems Review, Vol. 15, No. 5)
4. Rushby, J.: Separation and integration in MILS (The MILS Constitution). Technical report SRI-CSL-08-XX, February 2008
5. System and Network Analysis Center, Information Assurance Directorate, Separation Kernels on Commodity Workstations, NSA, 10 March 2010. https://www.niap-ccevs.org/announcements/Separation%20Kernels%20on%20Commodity%20Workstations.pdf
6. Schneider, F.B.: Beyond hacking: an SOS! In: Keynote at the 32nd ACM/IEEE International Conference on Software Engineering (ICSE), Cape Town, South Africa, p. 48 (2010). http://www.sbs.co.za/icse2010/_DOWNLOADS/Fred%20Schneider.pdf
7. Damgård, I.: A "proof-reading" of some issues in cryptography. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 2–11. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73420-8_2
8. Gligor, V.: Dancing with the adversary: a tale of wimps and giants. In: Christianson, B., Malcolm, J., Matyáš, V., Švenda, P., Stajano, F., Anderson, J. (eds.) Security Protocols 2014. LNCS, vol. 8809, pp. 100–115. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12400-1_11
9. In separation and integration in MILS, Technical report SRI-CSL-08-XX, February 2008, Rushby illustrates how to approximate information-flow properties with various safety properties. Hence, their composition by trace removal becomes fairly straight forward
10. This protocol is similar to that presented in Gligor, V.D., Woo, M.S.L.: Establishing software root of trust unconditionally. In: Proceedings of the Network and Distributed Systems Symposium (NDSS), San Diego, CA, February 2019 (Full paper: CMU - CyLab - Technical Report 18–003, November 2018)
11. Constantin, L.: What is a supply chain attack? In: Motherboard, 29 September 2017. https://motherboard.vice.com/en_us/article/d3y48v/what-is-a-supply-chain-attack
12. Lee, M., Moltke, H.: Everybody does it: the messy truth about infiltrating computer supply chains. The Intercept, 24 January 2019. https://theintercept.com/2019/01/24/computer-supply-chain-attacks
13. Arora, S., Barak, B.: Computational Complexity - A Modern Approach. Cambridge University Press, Cambridge (2009)
14. E.g., Figure 6 of Gligor, V.D., Woo, M.S.L.: Establishing software root of trust unconditionally. In: CMU - CyLab - Technical report 18-003, November 2018