A Security-Typed Language for Enforcing Non-Interference in Multi-Level Military Command and Control Systems

Moises John G. Adlawan., Reagan B. Ricafort

[1]School of Graduate Studies, AMA University, Quezon City, Philippines

DOI: https://doi.org/10.51584/IJRIAS.2026.11080058

Received: 19 August 2026; Accepted: 24 August 2026; Published: 05 September 2026

ABSTRACT

Military command and control systems handle data with different secrecy levels, even when the software runs in one shared networked setting. That mix can cause a security flaw. A program can be correct in how it works, but still let higher secrecy data affect lower secrecy results or messages. This work suggests ST-C2. It is a small programming language that uses security types. It uses three confidentiality tiers. Unclassified is L. Secret is M. Top Secret is H. ST-C2 puts several pieces together. It uses security labels. It uses a lattice style information flow rule. It checks security types before running code. It also labels the program counter, or PC. In addition, it uses communication channels that carry security labels.

To improve the earlier idea, the design is spelled out more clearly. It gives the core syntax. It defines state based operational rules. It states security typing judgments. It also states a termination insensitive non-interference rule for programs that pass the type checks. A proof-of-concept type checker was written in Python. The evaluation used 17 test cases. Some were meant to be secure. Others were meant to be insecure. The tests cover direct information moves. They also cover implicit moves that show up through PC labels. They include labeled communication channels as well.

In the end, the prototype handled all 17 cases the right way. On the chosen benchmark, the result was 100 percent. No false positives and no false negatives were seen. A basic timing check also matched an almost linear pattern for the small checking step as the number of checks rose. These results are only early. They do not replace a full compiler. They also do not act like a full operational military C2 system. Still, the study adds a formal language model aimed at C2 needs. It links well known programming language security tools to an explicit noninterference goal. It also sets a starting point for later work and implementation.

Keywords: Security-Typed Programming Language, Theory of Programming Languages, Information Flow Security, Non-Interference, Static Type Checking, Military Command and Control, Compiler.

INTRODUCTION

Modern Command and Control tools move data that comes with different security labels. This study looks at three labels. Unclassified is L. Secret is M. Top Secret is H. When data from these labels reaches other linked apps, the issue is bigger than who can log in. It is also bigger than which network is used or which operating system runs the software. The code itself can let data slip. This can happen if sensitive values end up changing a lower-level variable, a printed result, or a message path.

Good defenses still matter. Network split can help. Virtualization can help too. Role checks and operating-system protections also help. But these steps mostly guard the usual computer setup around the apps. They do not act like a built-in language rule that covers each data move caused by program logic. Language-based security is meant to close that gap. It ties limits on data movement to the type system and to the rules that give code its meaning (Sabelfeld & Myers, 2003; Smith, 2007).

A main goal of this study is non-interference. Put plainly, if data at a higher-level changes, the results seen at a lower level should not change. This setting looks at two kinds of influence. One is an explicit flow. For

example, publicData = missionData is a direct copy from high to low. The other is an implicit flow. This can happen when a secret value picks which path the program takes, and that choice then changes a public output. To track this, the approach uses program-counter labeling. The labels help carry the security level of the control flow into the checks for later statements (Volpano et al., 1996; Myers, 1999).

The theory rests on earlier work in several related areas. It builds on security lattices, information-flow models, non-interference, and ideas from secure type systems. Denning (1976) gives a lattice model for secure information flow. Bell and LaPadula (1973) offer a classic confidentiality model for labeled data. Goguen and Meseguer (1982) brought in non-interference as a tool for asking whether high-level actions can show up at low levels. Later studies tied these ideas to type systems in programming languages and to tools for static checking (Volpano et al., 1996; Myers, 1999; Sabelfeld & Myers, 2003).

This work does not claim a brand-new security theory. What it does is build a small language spec. It ties together a three-layer security lattice, program annotations, PC labels, labeled communication, and non-interference in a C2 focused language model. The first draft stayed at a high level. It did not include a clear grammar. It also lacked operational semantics, typing rules, and a working style demo. This version fixes those issues. It defines the core language in a formal way. It also adds a proof-of-concept type checker core.

The goal is to lay out the ST C2 language. Then it should spell out the syntax and the semantics. It also sets typing rules for both direct and indirect information flows. Next, it states a non-interference property. Finally, it shows the main rules using an executable prototype. So, the main value is a tighter Theory of Programming Languages style treatment of language based security for a multi-level C2 case. It is not a claim that each security mechanism is new on its own.

METHODOLOGY

Research Design

The study follows a design-and-formalization approach in Theory of Programming Languages. The work has four connected parts: (1) defining the security model, (2) specifying the language and its semantics, (3) formalizing the security typing rules and non-interference property, and (4) implementing and testing a proof-of-concept type-checking core. The prototype is deliberately limited to the security-checking mechanism and is not presented as a production compiler.

Security Model

ST-C2 uses a three-point confidentiality lattice with the ordering $L \sqsubseteq M \sqsubseteq H$. Information may flow from a lower level to an equal or higher level, but the core policy does not permit downward flows.

H(TopSecret)
|
M(Secret)
|
L (Unclassified)

The model is lattice-based in the sense of Denning (1976) and is compatible with the confidentiality direction of Bell-LaPadula. It is not claimed to be a complete Bell-LaPadula implementation because the present model focuses on programming-language information flow rather than the full access-control system.

Security Labels and Join Operation

Security labels are attached to variables, expressions, and communication channels. A program element is represented as an ordinary type paired with a security level, written τ^s. For example, missionData : Integer^H, unitStatus : String^M, and publicMessage : String^L. For compound expressions, the security level is the least upper bound (join) of the participating levels.

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VIII August 2026

L ⊔ L = L

L ⊔ M = M

M ⊔ H = H

L ⊔ H = H

Formal Language Syntax

The core language is intentionally small. It contains declarations, expressions, assignments, conditionals, loops, and communication channels. Functions, exceptions, concurrency primitives, dynamic objects, and declassification are outside the current core.

Program::=Declaration*Statement*

Declaration::=Level Identifier ":" Type";"

| Level "channel" Identifier";"

Level::="low" | "medium" | "high"

Type ::= "Integer" | "Boolean" | "String"

Statement ::= Identifier "=" Expression ";"

"if" "(" Expression ")" Block

| "if" "(" Expression ")" Block "else" Block

| "while" "(" Expression ")" Block

| "send" "(" Expression "," Identifier ")" ";"

Block ::= "{" Statement* "}"

Expression ::= Literal | Identifier

| Expression BinaryOp Expression

| UnaryOp Expression

| "(" Expression ")"

BinaryOp ::= "+" | "-" | "<" | ">" | "<=" | ">="

| "==" | "!=" | "&&" | "||"

UnaryOp ::= "!"

Literal ::= IntegerLiteral | StringLiteral | "true" | "false"

IntegerLiteral ::= digit+

StringLiteral ::= "" character* ""

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VIII August 2026

Operational Semantics

Let σ denote the program state, mapping variables to values. A transition $\langle C, \sigma\rangle \rightarrow \sigma'$ means that statement C executes in state σ and produces state σ′. For an assignment, the expression is evaluated and its value is stored in the destination:

$\langle x := e, \sigma\rangle \rightarrow \sigma[x \mapsto [\![e]\!]\sigma]$

For a conditional, the selected branch is executed according to the value of its condition. For a loop, the condition is evaluated before each iteration. For communication, the expression is evaluated and its value is sent through the named channel. These operational rules describe program behavior; whether an operation is security-safe is decided by the typing rules.

Non-Interference Property

The proof strategy consists of four main arguments: expression security, assignment safety, control-flow safety, and channel safety. Together they support preservation of Low-equivalence. A complete mechanized proof is outside the scope of the present prototype, so the theorem is presented with a proof strategy rather than as a machine-checked theorem.

Prototype Implementation and Evaluation Procedure

A proof-of-concept type-checking core was implemented in Python. It represents variables and communication channels with security labels and directly implements the lattice comparison, join operation, assignment checking, PC-label checking, and channel checking. It does not include a lexer, parser, abstract syntax tree, code generator, or complete compiler pipeline. The source file is supplied as ST-C2_Prototype_Type_Checker.py.

effective_level = join(source_level, pc)

if effective_level <= destination_level:

accept

else:

reject

The functional benchmark contains 17 deliberately constructed cases: six direct assignments, six PC-mediated cases, and five communication cases. Each case has a predetermined expected decision based on the formal rules. A case is counted as correct when the prototype decision matches the expected decision. False positives and false negatives are reported relative to this benchmark. A small processing-time check was also run by increasing the number of elementary assignment checks; the measurements are environment-dependent and are included only as preliminary scalability evidence

RESULTS

Functional Evaluation

Test Case	Expected	Prototype	Result
L → L assignment	Accept	Accept	Pass
L → M assignment	Accept	Accept	Pass

L → H assignment	Accept	Accept	Pass
M → M assignment	Accept	Accept	Pass
M → L assignment	Reject	Reject	Pass
H → L assignment	Reject	Reject	Pass
L PC → L write	Accept	Accept	Pass
M PC → M write	Accept	Accept	Pass
H PC → H write	Accept	Accept	Pass
M PC → L write	Reject	Reject	Pass
H PC → L write	Reject	Reject	Pass
H PC → M write	Reject	Reject	Pass
H data → H channel	Accept	Accept	Pass
M data → M channel	Accept	Accept	Pass
L data → H channel	Accept	Accept	Pass
M data → L channel	Reject	Reject	Pass
H data → L channel	Reject	Reject	Pass

All 17 deliberately constructed cases produced the expected decision. The benchmark accuracy was therefore 100% (17/17). There were zero observed false positives and zero observed false negatives within this selected benchmark. The result demonstrates consistency between the formal rules and the prototype implementation; it does not establish 100% detection accuracy for arbitrary programs.

Preliminary Processing-Time Check

Checked Operations	**Median Time**	**Approx. 95th Percentile**
10	0.0025 ms	0.0029 ms
100	0.0175 ms	0.0373 ms
1,000	0.1775 ms	0.2743 ms
5,000	0.9205 ms	2.6595 ms
10,000	1.9006 ms	2.9464 ms

The elementary checking operation showed increasing processing time as the number of checks increased. The measurements were approximately linear over the tested range. These values were obtained from the proof-of-concept checking core in the current execution environment; they are not hardware-independent compiler benchmarks and exclude parsing, code generation, file I/O, and other compiler activities.

Observed Security Behavior

The direct-flow tests accepted Low-to-Low, Low-to-Medium, Low-to-High, and Medium-to-Medium flows while rejecting Medium-to-Low and High-to-Low flows. The PC-mediated cases showed the same behavior when the program-counter label represented a sensitive control context. Communication tests rejected Medium or High information from being sent through a Low-level channel while permitting flows to channels at the same or higher level.

DISCUSSION

The findings link the ST-C2 language rules to what the model is supposed to do for security. The test results show that the prototype keeps the lattice order in place when it handles direct assignments, branch and loop code, and message paths. This matters because the earlier paper explained the ideas, but it did not show that the rules could turn into checks the system can run.

A label tied to the program counter matters a lot for implicit flows. With a direct assignment, it is usually clear what happened, since the secret tagged value sits on the right side of the statement. For an implicit flow, the sign is harder to spot. The tagged value may only affect which path the code takes. When the checker moves into the branch or the loop body, ST-C2 raises the PC level. That step makes the control choice itself part of the security rules. This matches the kind of method used in secure flow type systems (Volpano et al., 1996; Myers, 1999).

The formalization makes the non-interference goal easier to see. Instead of only saying that the language blocks leaks, it states Low-equivalence and then checks a termination-insensitive non-interference property. The assignment, control-flow, expression, and channel rules are laid out to support the claim that well-typed programs keep the same Low-level view. This links the security statement to syntax, meaning, and types, along with one named property. So, the contribution is a tighter Theory of Programming Languages result.

Still, this should be read as a proof of concept. The prototype is mainly a type-checking core. It is not a full compiler. It cannot parse whole ST-C2 programs, and it does not output runnable code. The benchmark uses only 17 cases, and the setup is rule-based. A perfect score here shows the system stays consistent on those cases, not that it gives broad security in general. The time measurements also cover only the basic checking step.

The proposed model also complements rather than replaces existing security controls. Network isolation, operating-system protection, authentication, access control, and infrastructure safeguards remain necessary in a real C2 environment. ST-C2 addresses a different layer by making certain information-flow restrictions part of the programming language itself.

Declassification and Future Extensions

The core intentionally keeps declassification out. This careful choice keeps the rule that data only moves in one direction. Real software may still need a permitted release of some facts. When that happens, the release should show up as a clear language form. It should follow set policy rules. It also needs logs and checks. Otherwise, letting normal assignment skip the security rules would be wrong (Sabelfeld & Sands, 2009).

Another point is concurrency. When threads run together, timing and shared state can leak data in new ways. If memory is dynamic, labels have to follow values through new allocations. Labels also need to stay correct for pointers and aliases. With distributed systems, messages cross nodes. So the model must cover more than one endpoint. It must also handle separate security areas. A bigger lattice may help too. That could fit systems with more than a few security groups. Next, the build plan should add a lexer and a parser. After that, it should create an abstract syntax tree. Then a full type checker for security is needed. Last, add an executable target or an intermediate code back end. The test set should be larger as well. It should include deep if cases, while loops, and arithmetic. Mix of levels also matters. Add communication cases too. Try combinations of these features, not only one at a time.

Later evaluation should spell out several metrics. Report detection quality. Give false positive and false negative rates. Include any checking or compile time costs. Also test how it scales in a setup that is written down and easy to repeat. For stronger theory, aim for a machine-checked non-interference proof.

Limitations

• The current language is intentionally small and does not include functions, exceptions, concurrency, dynamic memory, or a full module system.

• The security lattice contains only three levels and does not model additional compartments or categories.

• The non-interference property is termination-insensitive and assumes both executions terminate.

• The prototype is a proof-of-concept type-checking core rather than a complete compiler.

• The functional evaluation uses deliberately constructed cases rather than complete C2 application programs.

• Declassification is not included in the current core language.

• The timing measurements are environment-dependent and should be repeated under a controlled hardware and software configuration for a stronger performance study.

CONCLUSION

This paper introduces ST-C2, a programming language with security types. It is meant to guide how data moves in multi-level military Command and Control software. The new version does more than outline the idea. It also sets out a three-step confidentiality lattice. It gives a basic grammar and operational semantics. It includes typing judgments and program-counter labeling. It covers labeled communication too. It then states a termination-insensitive non-interference result.

The rules cover two kinds of data movement. One kind is explicit flow. The other is implicit flow. You cannot put a high value into a low target. The reason is that H is not allowed to flow to L in the lattice order. You also cannot let a high test quietly steer a low write. During checking, the program-counter label is raised, so the low write is not treated as safe. The same idea shows up in the communication rule. Channels carry security labels. If a sender and receiver labels do not match the allowed flow, the type system blocks the action.

In the test run, the type checker handled all 17 cases that were set up on purpose. It hit 100% on the benchmark. I also did not see any false positives or false negatives. I ran a quick timing check too. For the basic checking step, the time rose in a near straight line as the number of checks went up. Still, this does not prove it is ready for use in a real military setting.

ST-C2 mainly aims at bringing known secure programming language ideas into a small, C2 focused language design. It gives a clearer grounding in programming language theory for this task. It also offers a practical base for the next work. That next work includes building a fuller compiler, doing formal checks, and expanding the security tests.

REFERENCES

1. Bell, D. E., & LaPadula, L. J. (1973). Secure computer systems: Mathematical foundations. MITRE Corporation.
2. Denning, D. E. (1976). A lattice model of secure information flow. Communications of the ACM, 19(5), 236–243. https://doi.org/10.1145/360051.360056
3. Goguen, J. A., & Meseguer, J. (1982). Security policies and security models. In 1982 IEEE Symposium on Security and Privacy (pp. 11–20). IEEE.

4. Myers, A. C. (1999). JFlow: Practical mostly-static information flow control. In Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (pp. 228–241). ACM. https://doi.org/10.1145/292540.292561
5. Pierce, B. C. (2002). Types and programming languages. MIT Press.
6. Russo, A., & Sabelfeld, A. (2010). Dynamic vs. static flow-sensitive security analysis. In 2010 IEEE 23rd Computer Security Foundations Symposium (pp. 186–199). IEEE. https://doi.org/10.1109/CSF.2010.21
7. Sabelfeld, A., & Myers, A. C. (2003). Language-based information-flow security. IEEE Journal on Selected Areas in Communications, 21(1), 5–19. https://doi.org/10.1109/JSAC.2002.806121
8. Sabelfeld, A., & Sands, D. (2009). Declassification: Dimensions and principles. Journal of Computer Security, 17(5), 995–1023. https://doi.org/10.3233/JCS-2009-0353
9. Smith, G. (2007). Formal models of information flow. In Foundations of Security Analysis and Design V (pp. 1–43). Springer. https://doi.org/10.1007/978-3-540-75227-9_1
10. Volpano, D., Smith, G., & Irvine, C. (1996). A sound type system for secure flow analysis. Journal of Computer Security, 4(2–3), 167–187. https://doi.org/10.3233/JCS-1996-42-305